NPAC Technical Report SCCS–636, Sept. 1994
Also available as CRPC Technical Report CRPC–TR94483–S

PASSION: Parallel And Scalable Software for Input-Output *

Alok Choudhary *Rajesh Bordawekar* *Michael Harry* *Rakesh Krishnaiyer*
Ravi Ponnusamy *Tarvinder Singh* *Rajeev Thakur*

ECE Dept., NPAC and CASE Center,
Syracuse University, Syracuse, NY 13244
choudhar, rajesh, mharry, rakesh, ravi, tpsingh, thakur @npac.syr.edu

Abstract

We are developing a software system called **PASSION**: **P**arallel **A**nd **S**calable **S**oftware for **I**nput-**O**utput which provides software support for high performance parallel I/O. PASSION provides support at the language, compiler, runtime as well as file system level. PASSION provides runtime procedures for parallel access to files (read/write), as well as for out-of-core computations. These routines can either be used together with a compiler to translate out-of-core data parallel programs written in a language like HPF, or used directly by application programmers. A number of optimizations such as Two-Phase Access, Data Sieving, Data Prefetching and Data Reuse have been incorporated in the PASSION Runtime Library for improved performance. PASSION also provides an initial framework for runtime support for out-of-core irregular problems. The goal of the PASSION compiler is to automatically translate out-of-core data parallel programs to node programs for distributed memory machines, with calls to the PASSION Runtime Library. At the language level, PASSION suggests extensions to HPF for out-of-core programs. At the file system level, PASSION provides support for buffering and prefetching data from disks. A portable parallel file system is also being developed as part of this project, which can be used across homogeneous or heterogeneous networks of workstations. PASSION also provides support for integrating data and task parallelism using parallel I/O techniques.

We have used PASSION to implement a number of out-of-core applications such as a Laplace's equation solver, 2D FFT, Matrix Multiplication, LU Decomposition, image processing applications as well as unstructured mesh kernels in molecular dynamics and computational fluid dynamics. We are currently in the process of using PASSION in applications in CFD (3D turbulent flows), molecular structure calculations, seismic computations, and earth and space science applications such as Four-Dimensional Data Assimilation. PASSION is currently available on the Intel Paragon, Touchstone Delta and iPSC/860. Efforts are underway to port it to the IBM SP-1 and SP-2 using the Vesta Parallel File System.

*This work was supported in part by NSF Young Investigator Award CCR-9357840, grants from Intel SSD and IBM Corp., in part by USRA CESDIS Contract # 5555-26 and also in part by ARPA under contract # DABT63-91-C-0028. The content of the information does not necessarily reflect the position or the policy of the US Government and no official endorsement should be inferred. Rajeev Thakur is supported by a Syracuse University Graduate Fellowship. Michael Harry is supported by an ARPA Assert Fellowship. Rakesh Krishnaiyer is supported by the CASE Center, a NY State Advance Technology Center. This work was performed in part using the Intel Touchstone Delta and Paragon Systems operated by Caltech on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by CRPC. This work was also performed in part using the Intel iPSC/860 and IBM SP-1/SP-2 at NPAC, the IBM SP-1 at Argonne National Laboratory and the Intel Paragon at the Jet Propulsion Laboratory.

Contents

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. A large number of applications in diverse areas such as large scale scientific computations, database and information processing, hypertext and multimedia systems, information retrieval etc. require processing very large quantities of data. For example, a typical Grand Challenge Application at present 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-2, nCUBE2 etc. provide some kind of hardware and software support for parallel I/O [CFPB93, Pie89, DdR92].

At Syracuse University, we are investigating the I/O problem from a software perspective, including languages, compilers, runtime support and file system optimizations. The overall project is called **PASSION** which stands for **P**arallel **A**nd **S**calable **S**oftware for **I**nput-**O**utput. PASSION provides support for compiling out-of-core data parallel programs [TBC94a, BCT94], parallel input-output of data and parallel access to files [BdRC93], communication of out-of-core data, redistribution of data stored on disks, many optimizations including Data Prefetching from disks, Data Sieving, Data Reuse etc., as well as support at the file system level. We have also developed an initial framework for runtime support for out-of-core irregular problems.

This report gives an overview of the design and implementation of the various components of PASSION.

2 PASSION Overview

PASSION provides software support for I/O intensive loosely synchronous problems [FWM94]. It has a layered approach and provides support at the compiler, runtime support and file system levels as shown in Figure 1. The various components of PASSION are briefly explained below and described in more detail in later sections. Further details may be obtained from papers and reports listed in Section 11.

1. **Parallel Access to Files:** PASSION provides support for parallel access to files for read/write operations, supports distribution of data on parallel file systems as well as distribution and redistribution of data among the processors of a distributed memory machine. It uses the Two-Phase Access Strategy [dRBC93, BdRC93] for this purpose. Section 3 describes this work in further detail.

2. **Out-of-Core Computations:** Out-of-Core Computations are those in which the primary data structures are too large to fit in main memory and hence reside on disks. PASSION provides runtime, compiler and language support for these computations. Section 4 describes this work in further detail.

3. **File System Optimizations:** One of the key components of PASSION is to use the access pattern knowledge extracted from the program and provided by the compiler and runtime system to the file system, so that certain optimizations can be performed. These optimizations include efficient management and allocation of caches and buffers, scheduling of I/O accesses to disks and prefetching from disks based on the access patterns. Figure 2 shows how information is passed from the program to the compiler, from the compiler to the runtime system and from the compiler to the file system. Section 6 describes this work in further detail.

4. **VIP-FS Portable Parallel File System:** VIP-FS provides a portable parallel file system in a distributed computing environment. The file system is deemed a *virtual* file system because it is implemented using multiple individual standard file systems integrated by a message passing system. VIP-FS is portable across many architectures as well as many message passing systems and is designed to work in a heterogeneous environment. Section 7 describes this work in further detail.

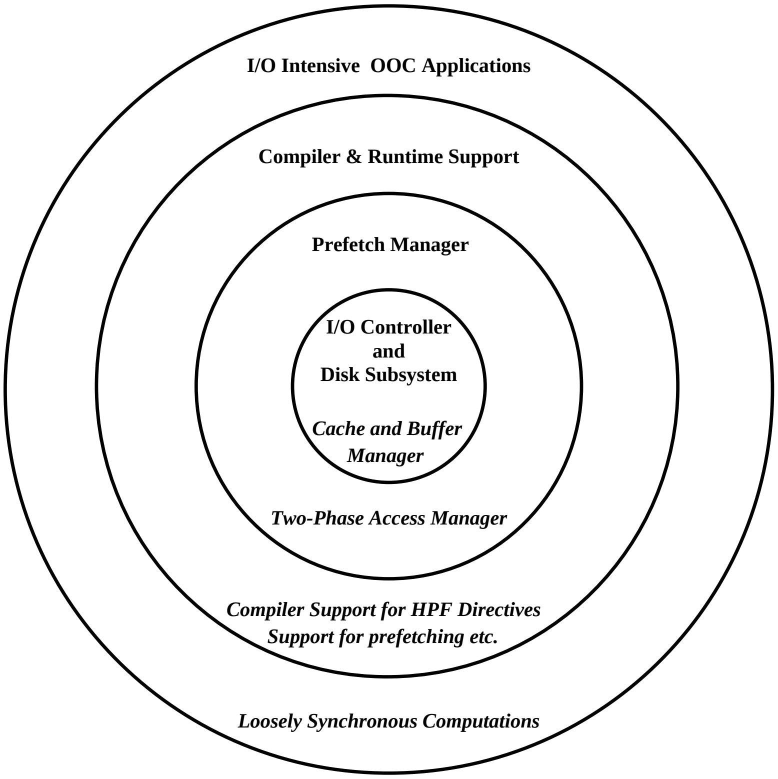

I/O Intensive OOC Applications
Compiler & Runtime Support
Prefetch Manager
I/O Controller
and
Disk Subsystem
Cache and Buffer
Manager
Two-Phase Access Manager
Compiler Support for HPF Directives
Support for prefetching etc.
Loosely Synchronous Computations

Figure 1: PASSION Rings

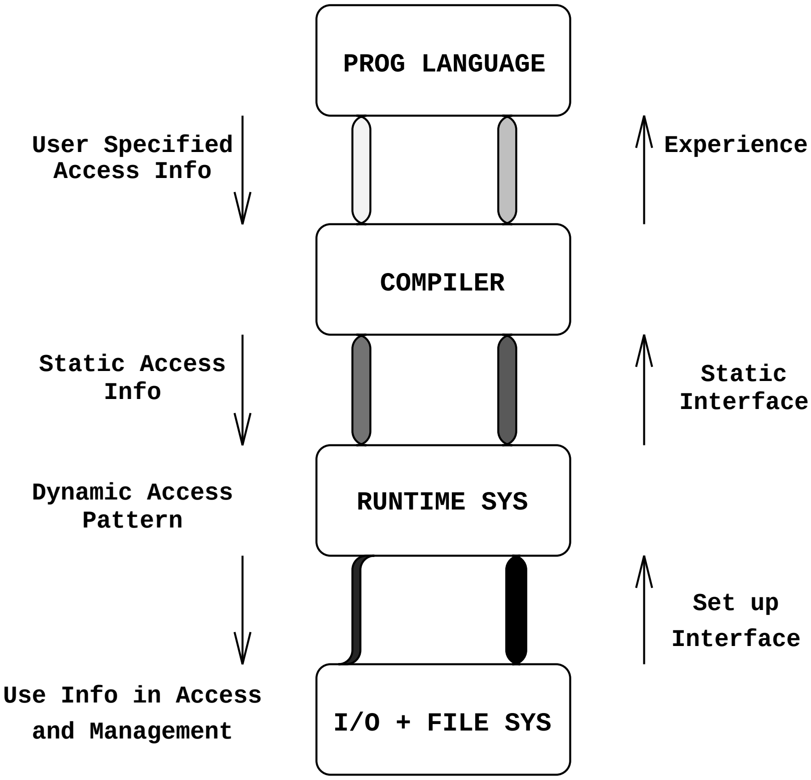


Figure 2: Information Flow in PASSION

5. **Integrating Task and Data Parallelism Using Parallel I/O:** This component deals with providing a "parallel pipes" mechanism for communication among data parallel tasks. Since data distributions in two data parallel tasks may be different and may not be known in advance, this component provides a way to perform communication in parallel while hiding the individual distributions, and redistributing the data if needed. Section 8 describes this work in further detail.

3 Parallel Access to Files

Many applications require accessing large arrays from disks and distributing them among the processors in some fashion. In order to obtain better I/O performance, data is usually stored in a parallel file system in which the file is striped across a number of disks. This enables the file to be read in parallel by many processors. The performance of parallel read and write operations depends to a large extent on the way data is distributed on disks and processors. Data distribution on disks depends the data striping method and file storage order (row-major or column-major). The data distribution on the processors is said to be a *conforming* distribution if it results in accessing consecutive data blocks from files. It has been observed that I/O performance is very good in the case of conforming distributions. Other data distributions give much lower performance. To alleviate this problem, the **Two Phase Access Strategy** has been proposed in [Bor93, BdRC93, dRBC93]. This strategy is an alternative to accessing data directly according to the data distribution (called **Direct Access Strategy**).

3.1 Two-Phase Access Strategy

In the Two Phase Approach, data is first read in a manner conforming to the distribution on disks and then redistributed among the processors to obtain the target distribution. The data access cost can be now computed as a sum of file access cost and data redistribution cost. The file access cost for a conforming distribution is fixed for a particular machine and programming language. Since the redistribution cost is very small as compared to the file access cost, the cost of accessing data becomes independent of the data distribution on the disks. This is found to give consistently good performance for all distributions [Bor93, BdRC93, dRBC93]. The Two-Phase Approach provides the following advantages over the conventional Direct Access Method:-

1. The distribution of data on disks is effectively hidden from the user.
2. It uses the higher bandwidth of the interconnection network.
3. It uses collective communication and collective I/O operations.
4. It provides software caching of the out-of-core data in main memory to exploit temporal and spatial locality.
5. It aggregates I/O requests of compute nodes so that only one copy of each data item is transferred between disk and main memory.

Figure 3 shows the performance for reading a 10K × 10K array on the Intel Touchstone Delta using 64 processors. For Fortran programs the column-block distribution is the conforming distribution, so for this distribution the Two-Phase and Direct Access Methods take the same time. For all other distributions, data is first accessed assuming a column block distribution and then redistributed to obtain the target distribution. This approach gives considerable better performance than accessing data using the **Direct Access Method**.

3.2 PASSION Routines for Accessing Files

PASSION provides runtime routines for reading/writing data from parallel files for any kind of data distribution. The Two-Phase Access Strategy is used for this purpose. These routines can be called from an HPF or Fortran 77 program. PASSION also provides a set of auxiliary routines which store information in data structures about distributed arrays, data files, processor configuration etc. This information is then used by the file access routines. Some of the file access and auxiliary routines are listed in Table 1.

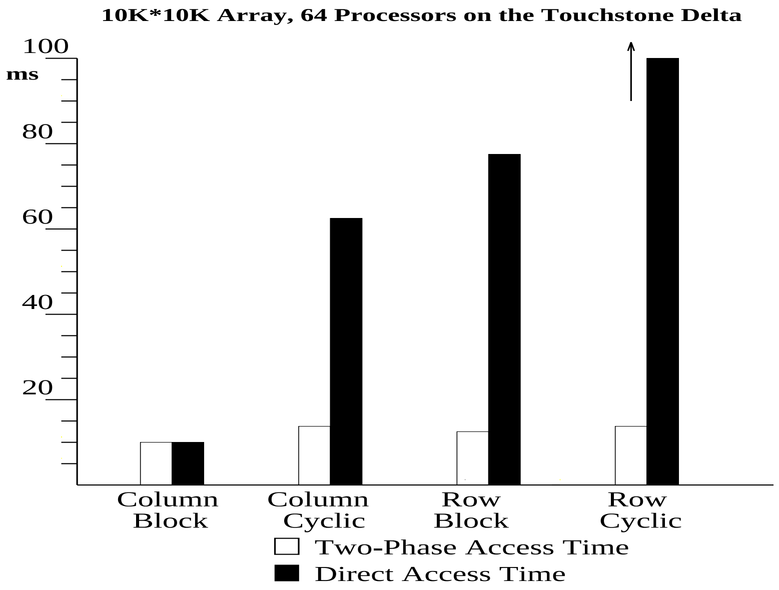


Figure 3: Two-Phase v/s Direct Access Method

Table 1: PASSION Routines for Parallel File Access

	Auxiliary Routines	
	PASSION Routine	Function
1	**PASSION_array_map**	Store information about distributed arrays
2	**PASSION_file_map**	Store information about files
3	**PASSION_proc_map**	Store information about the processor configuration
	File Access Routines	
	PASSION Routine	Function
1	**PASSION_read**	Read a distributed array from file
2	**PASSION_write**	Write a distributed array to file

4 Out-of-Core Computations

4.1 Models

This section discusses the architectural, programming and data storage models used by PASSION for out-of-core computations.

4.1.1 Architectural Model

An important goal in the design of PASSION has been to make it architecture independent as far as possible. The architectural model assumed by PASSION is that of any general distributed memory computer in which the processors are connected together in some fashion. The system is assumed to be provided with a set of disks and I/O nodes. The I/O nodes can either be dedicated processors or some of the compute nodes may also serve as I/O nodes. Each processor may either have its own local disk or all processors may share the set of disks. The I/O subsystem may have a separate interconnection network or it can share the same network which connects the processors together. The I/O routines in PASSION have been implemented using the native parallel file system calls provided on the machine. Hence PASSION can be easily ported to different machines by modifying only a few of its routines, mainly those which perform I/O and communication. PASSION is currently available on the entire family of Intel computers, namely the Touchstone Delta, Paragon and iPSC/860 using the Concurrent File System (CFS) and efforts are underway to port it to the IBM SP-1 and SP-2 using the Vesta Parallel File System and the VIP-FS File System (Section 7).

4.1.2 Data Storage and Access Models

Since PASSION is used in programs having large arrays which do not fit in main memory, the arrays have to stored on disks in some fashion. PASSION supports three basic models of storing and accessing arrays, called the Local Placement Model (LPM), the Global Placement Model (GPM) and the Partitioned-Incore Model (PIM). The PASSION runtime system automatically handles the input-output of data for each of these models, transparent to the user.

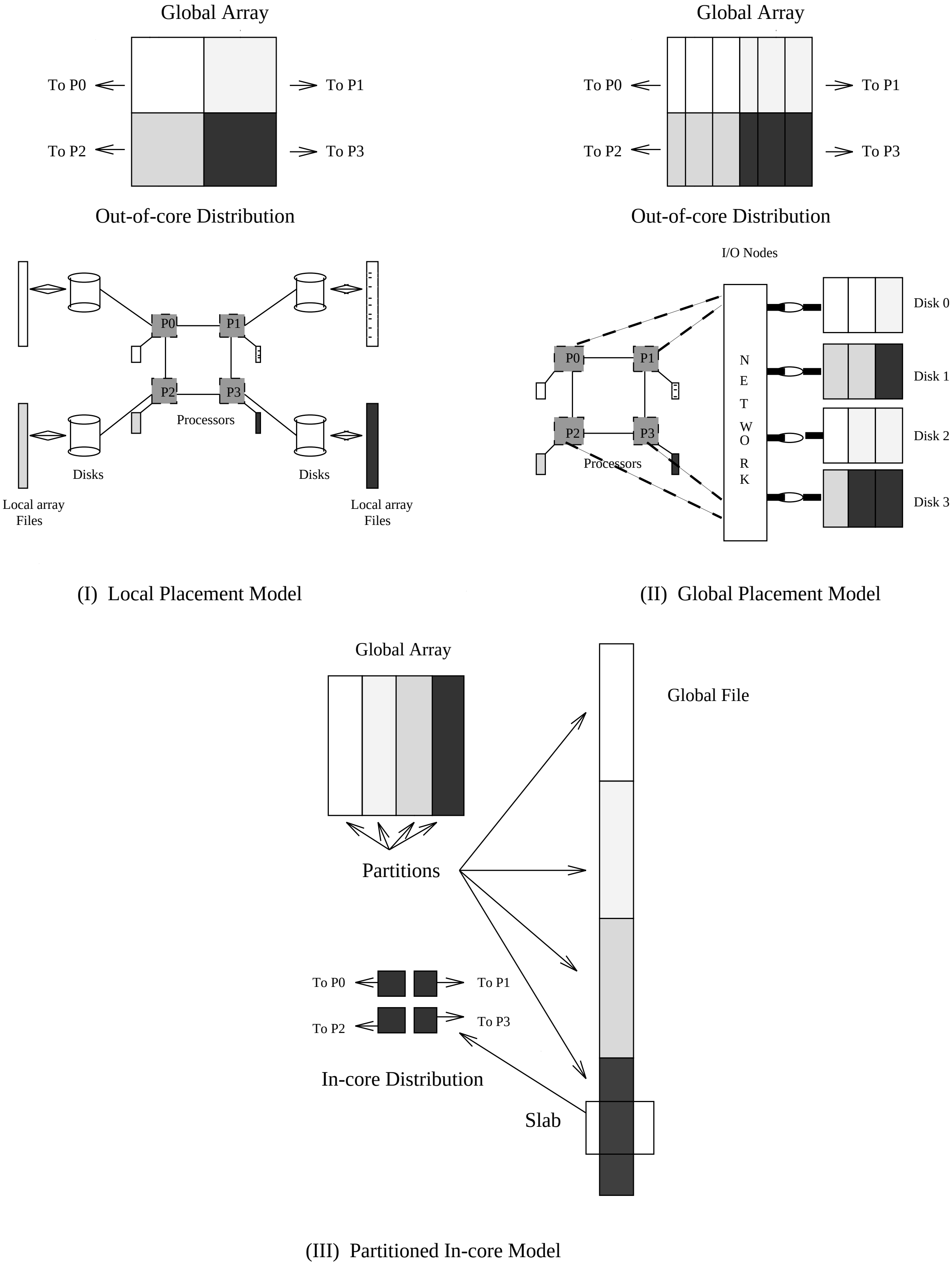
Local Placement Model (LPM): In this model, the global array is divided into local arrays belonging to each processor. Since the local arrays are out-of-core, they have to be stored in files on disk. The local array of each processor is stored in a separate file called the **Local Array File (LAF)** of that processor as shown in Figure 4(I). The node program explicitly reads from and writes into the file when required. 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. 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 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.

Global Placement Model (GPM): In this model, the global array is stored in a single file called the **Global Array File (GAF)** as shown in Figure 4(II) and no local array files are created. The global array is only logically divided into local arrays in keeping with the SPMD programming model. But, there is a single global array on disk. The PASSION runtime system automatically fetches the appropriate portion of each processor's local array from the global array file. The advantage of the Global Placement Model is that it saves the initial local array file creation phase in the local array model. The disadvantage is that each processor's data may not be stored contiguously in the GAF, resulting in higher I/O latency time. Also, explicit synchronization is required when a processor needs to access data belonging to another processor.

Partitioned-Incore Model (PIM): The Partitioned-Incore Model illustrated in Figure 4(III) is a variation of the Global Placement Model. The array is stored in a single global array file as in the Global Placement Model, but there is a difference in the way data is accessed. In the Partitioned-Incore Model, the global array is logically divided into a number of *partitions*, each of which can fit in the main memory of all processors combined. Thus the computation on each partition is essentially an in-core problem and no I/O is required during the computation on the partition. Hence the name Partitioned-Incore Model. The reading of each partition and its in-core distribution among processors is done by the PASSION runtime system using the Two-Phase Data Access Method [BdRC93, dRBC93]. This model is useful when the data access pattern in the program has good locality. Otherwise, creating in-core partitions itself is difficult.

4.2 Runtime Support

In out-of-core computations, data needs to be moved back and forth between main memory and disks. Also, 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, runtime support is needed to perform I/O as well as communication. The PASSION Runtime Library for out-of-core computations 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



(I) Local Placement Model

(II) Global Placement Model

(III) Partitioned In-core Model

Figure 4: Data Storage Models

Table 2: Some of the PASSION Routines for Out-of-core Computations

	Array Management Routines	
	PASSION Routine	Function
1	**PASSION_read_section**	Read a regular section from LAF to ICLA
2	**PASSION_write_section**	Write a regular section from ICLA to LAF
3	**PASSION_read_with_reuse**	read_section with data reuse [TBC94a]
4	**PASSION_prefetch_read**	Asynchronous (non-blocking) read of a regular section
5	**PASSION_prefetch_wait**	Wait for a prefetch to complete
	Communication Routines	
	PASSION Routine	Function
6	**PASSION_oc_shift**	Shift type collective communication on out-of-core data
7	**PASSION_oc_multicast**	Multicast communication on out-of-core data
	Mapping Routines	
	PASSION Routine	Function
8	**PASSION_oc_disk_map**	Map disks to processors
9	**PASSION_oc_file_map**	Generate local files from global files
	Generic Routines	
	PASSION Routine	Function
10	**PASSION_oc_transpose**	Transpose an out-of-core array
11	**PASSION_oc_matmul**	Perform out-of-core matrix multiplication

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.

4.2.1 Out-of-Core Runtime Library

The PASSION routines for out-of-core computations can be divided into four main categories based on their functionality — Array Management/Access Routines, Communication Routines, Mapping Routines and Generic Routines. Some of the basic routines and their functions are listed in Table 2.

1. **Array Management/Access Routines:** These routines handle the movement of data between the arrays in main memory and files on disks. Any arbitrary regular section of the 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) [TBC94a]. The Data Sieving Method described in Section 4.5.1 is used for improved performance.

2. **Communication Routines:** The Communication Routines perform collective communication of data in the OCLA. We use the Explicit Communication Method described in [TBC94a]. 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. **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. Disk mapping routines map physical disks onto logical disks.

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.

4.3 Runtime Support for Out-of-Core Structured Problems

```
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 5: HPF Program Fragment

Consider the HPF program fragment shown in Figure 5, 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 6. Consider the out-of-core local array on processor P5, shown in Figure 6(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 6(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 6(D). Data from the file is read into 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, this can be done by either directly reading other processors' LAFs (Direct File Access Method), or by using the Two-Phase Approach described earlier.

4.3.1 Performance Results

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 given in Table 3. 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 — shift using direct file access, shift using Two Phase Method and shift using two phase with data reuse, an optimization described in Section 4.5.3. 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 two phase method with data reuse as it reduces the amount of I/O by reusing 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

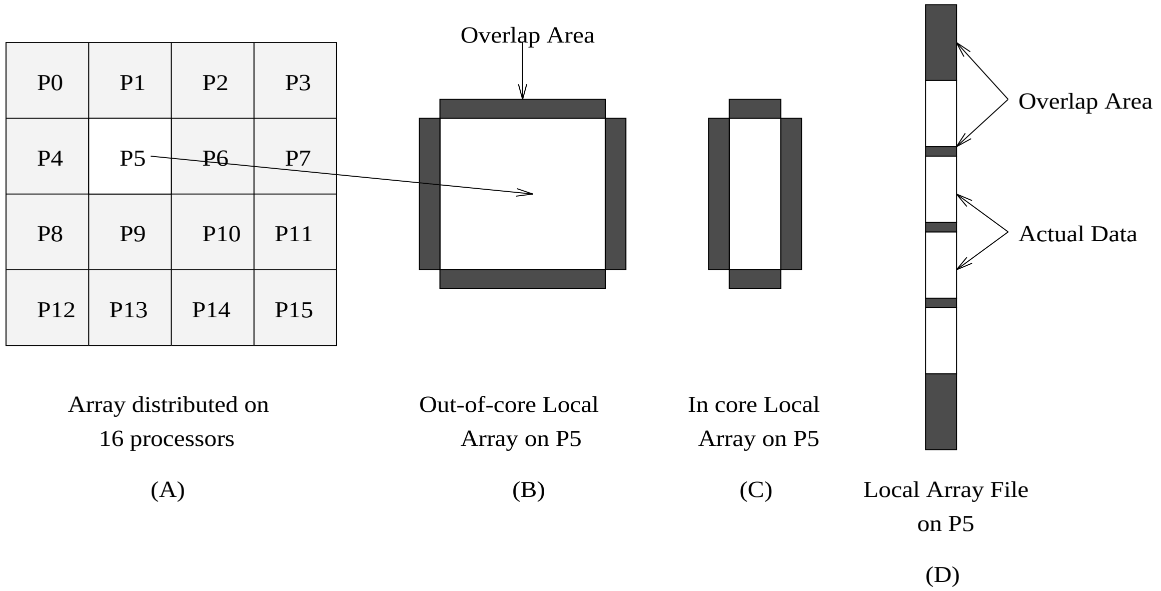


Figure 6: Example of OCLA, ICLA and LAF

Table 3: 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
Shift using Direct File Access	73.45	79.12	265.2	280.8
Shift using Two Phase	68.84	75.12	259.2	274.7
Shift using Two Phase with data reuse	62.11	71.71	253.1	269.1

a disk when they try to read the boundary values.

Table 4 presents the performance of the two-dimensional Fast Fourier Transform on the Intel Touchstone Delta. The Fast Fourier Transform routine uses the PASSION routine `PASSION_oc_transpose` 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 row slabs. Then the out-of-core array is transposed and the in-core FFT is again performed on the row 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 decreases. 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.

4.4 Runtime Support for Out-of-Core Unstructured Problems

Unstructured or Irregular problems [Pon94] are an important subclass of scientific applications. In irregular problems, the data access patterns cannot be predicted until runtime. Hence, 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 perform optimizations. Preprocessing methods for in-core computations have been developed [MSS+88, Pon94] for a variety of unstructured problems including explicit multigrid 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 7 illustrates a typical irregular loop. The data access pattern is determined by the array **indir**,

Table 4: Out-of-core 2D-FFT for 4K × 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

```
        real     x(n_node),y(n_node)                         ! data arrays
        integer indir(n_edge,2)                              ! indirection array

L1:     do i = 1, n_step                                     ! outer loop
L2:       do j = 1, n_edge                                   ! inner loop
            x(indir(i,1)) = x(indir(i,1)) + y(indir(i,2))
            x(indir(i,2)) = x(indir(i,2)) + y(indir(i,1))
          end do
        end do
```

Figure 7: An Example with an Irregular Loop

which is known only at runtime. This array is called an *indirection array*. Prior knowledge of loop data access patterns (values of **indir**) makes it possible to predict which data elements need to be communicated between processors. By pre-processing data access patterns, optimizations such as *software caching* and *communication vectorization* [DMS$^+$94] can be performed. We have developed runtime routines in PASSION to solve out-of-core irregular problems on distributed memory machines. The out-of-core pre-processing phase analyzes data access patterns and computes a communication schedule [MSS$^+$88] as well as an I/O schedule.

We describe our scheme using a computation kernel similar to the loop shown in Figure 7, on an unstructured mesh. Such a computation forms the core of many applications in fluid dynamics, molecular dynamics etc. The calculation on each node of the mesh requires data from its neighboring nodes. We assume that the size of both data and indirection arrays is very large and that only one of them can be stored in main memory, while the other has to be stored on disks. This out-of-core unstructured computation is done in three stages namely: data and/or indirection array partitioning, pre-processing of the indirection array and actual computation using the information provided by the previous two stages. Some of the main functions used are listed in Table 5.

4.4.1 Data/Indirection Array Partitioning

In this phase, the data and/or indirection array is divided into smaller partitions which can fit in the main memory of the processors [Figure 8]. This is done using a graph partitioning algorithm such as Recursive Coordinate Bisection, so as to maintain data locality as well as load balance. In the present implementation, we partition either the data or the indirection arrays but not both, i.e. one of them is assumed to be in-core and the other out-of-core. The partitioning is done in either of the following ways, depending on the model of data storage and access. This is illustrated in Figure 8.

- N **Partitioning:** This corresponds to the Local Placement Model. The data (or indirection) array is partitioned into some N independent partitions, each of which can fit in the main memory of any one processor.
- **Hierarchical Partitioning:** This corresponds to the Partitioned In-Core Model. Partitioning is done in two levels. In the first level, the data (or indirection) array is divided into a certain number of partitions, each of which can fit the main memory of all processors combined. In the second level, each of the partitions created above is further divided into as many partitions as the number of processors.

Table 5: PASSION Runtime Routines for Out-of-Core unstructured problems

	Data/Indirection Array Partitioning	
	PASSION Routine	Function
1	**PASSION_oc_gen_partition**	Partition data/indirection array
	Pre-Processing Procedures	
	PASSION Routine	Function
2	**PASSION_oc_dereference**	Translate indices for data/indirection arrays
3	**PASSION_oc_gen_pal**	Generate partition allocation list for each partition
4	**PASSION_oc_localize**	Out-of-core localization for off-processor array elements.
5	**PASSION_oc_gen_schedule**	Generate schedule for all the partitions.
	Computation Procedures	
	PASSION Routine	Function
6	**PASSION_oc_get_partition**	Load partition from disk
7	**PASSION_oc_gather**	Gather off-partition elements from memory and disks
8	**PASSION_oc_scatter**	Scatter off-partition elements from memory and disks.

4.4.2 Pre-Processing

This phase performs the following steps:-

- If the data (or indirection) array has been partitioned in the previous phase, the indirection (or data) array is divided into the same number of partitions. An indirection (or data) array element is assigned to a particular data (or indirection) partition if most of the data (or indirection) elements required in that iteration are in that partition. If the new partitions formed are larger than the available memory, they are processed one slab at a time.

- For each partition, a list of data (or indirection) elements required from some other partition is created. This list is important in optimizing the communication between processing nodes and disks. It also determines the size of the extra buffer space needed for out-of-partition data for computation on each partition. Localization and schedule generation procedures are used for this step.

- Perform address translation for references to out-of-partition elements [DMS$^+$94].

4.4.3 Computation

If there are P processing nodes, the computation involves reading in P partitions at a time from disks, evaluating partial results and storing them back on disks [Figure 9]. This process is repeated for all partitions and partial results are gathered. Final results are scattered to the partitions on other processors or files on disk.

4.4.4 Performance Results

We studied the performance of the out-of-core unstructured molecular dynamics and CFD codes on an Intel iPSC/860 with 16 processing nodes, 2 I/O nodes and 4 disks. In the molecular dynamics code, the data array is small, but the indirection array is very large. Hence we kept the indirection array out-of-core and the data array in-core.

Tables 6 and 7 show the pre-processing time for the cases with 14K and 6K atoms respectively. The slab size as well as number of processors was varied. It can be observed that as the slab size is increased, the performance improves because the number of I/O accesses is reduced. The performance also improves as the number of processors is increased. The best performance is observed for 8 processors with slab size of 64K. The performance degrades for the 16 processor case because the I/O subsystem gets saturated.

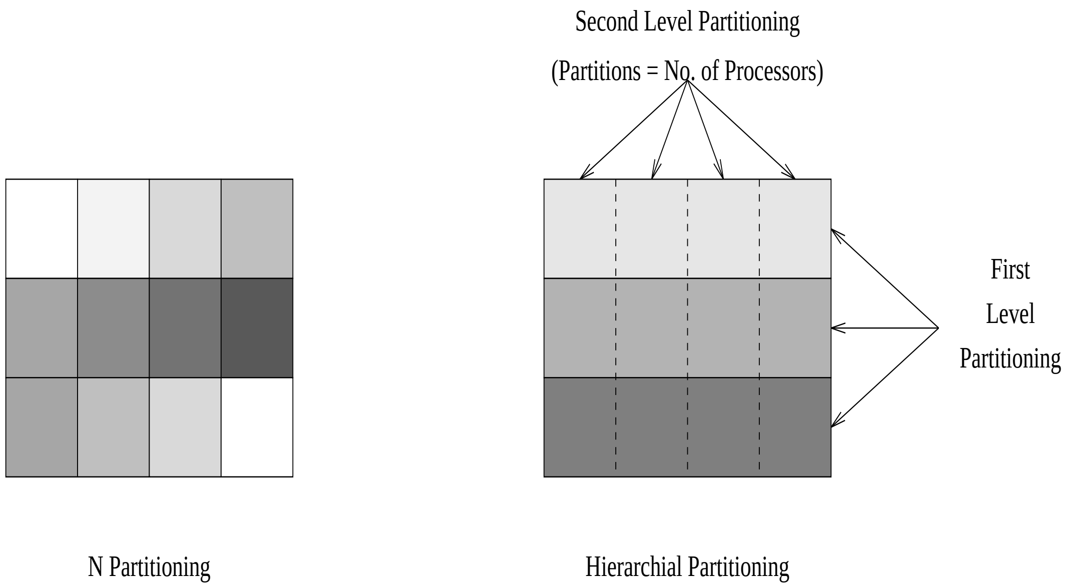


Figure 8: Data/Indirection Array Partitioning

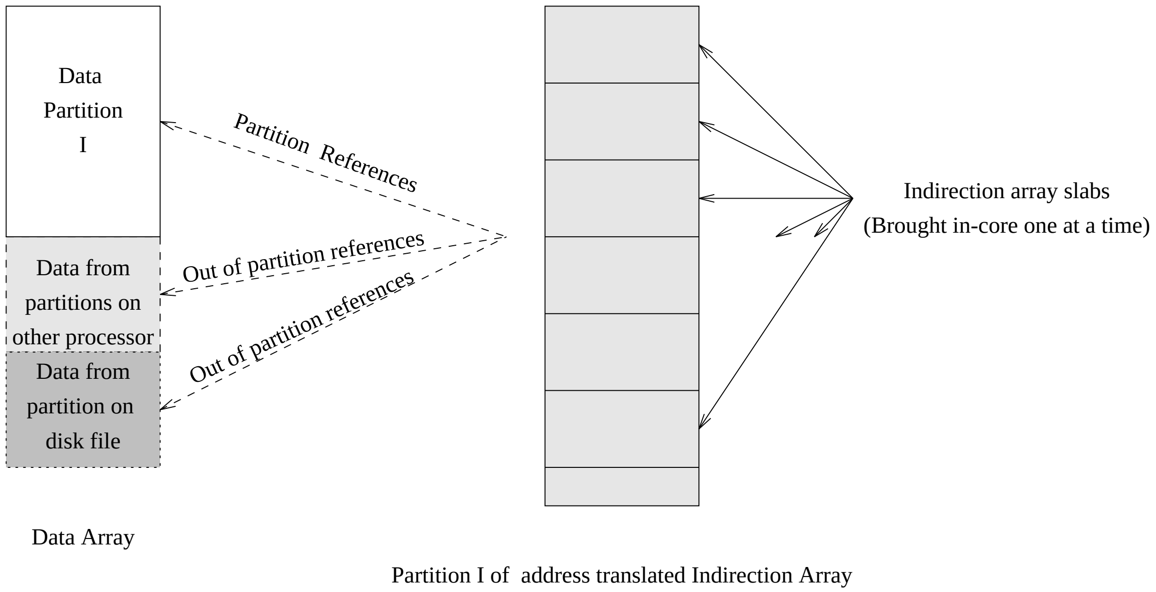

Data
Partition
I
Data from
partitions on
other processor
Data from
partition on
disk file
Partition References
Out of partition references
Out of partition references
Indirection array slabs
(Brought in-core one at a time)
Data Array
Partition I of address translated Indirection Array

Figure 9: Computation Phase: Data partition is indirectly referenced using indirection array partition

Table 6: Pre-processing Time in sec. for MD Kernel (14K Atoms)

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 7: Pre-processing Time in sec. for MD Kernel (6K Atoms)

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

Table 8: Computation Time in sec. for MD Kernel (14K Atoms)

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

Tables 8 and 9 show the computation times for one iteration of the molecular dynamics kernel for various number of processors and slab sizes. This time also includes the communication time to gather and scatter off-processor data. The best performance is observed for 8 processors in the 14K atoms case and for 4 processors in the 6K atoms case.

For the unstructured CFD problem, we used the N Partitioning Scheme on the data arrays. The pre-processing time and computation time for a 50K mesh are shown in Figures 10 and 11 respectively. As the number of processors is increased, both pre-processing and computation times decrease. Also, the performance is better for smaller number of partitions (ie. large partition size). This is because of the smaller number of I/O requests.

4.5 Optimizations

This section describes some of the optimizations incorporated in the PASSION Runtime Library for out-of-core computations.

4.5.1 Data Sieving

All the PASSION runtime routines for reading or writing data from/to disks support the reading/writing of regular sections of arrays. We define a *regular section* of an array as any portion of an array which can be specified in terms of its lower bound, upper bound and stride in each dimension. The need for reading

Table 9: Computation Time in sec. for MD Kernel (6K Atoms)

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

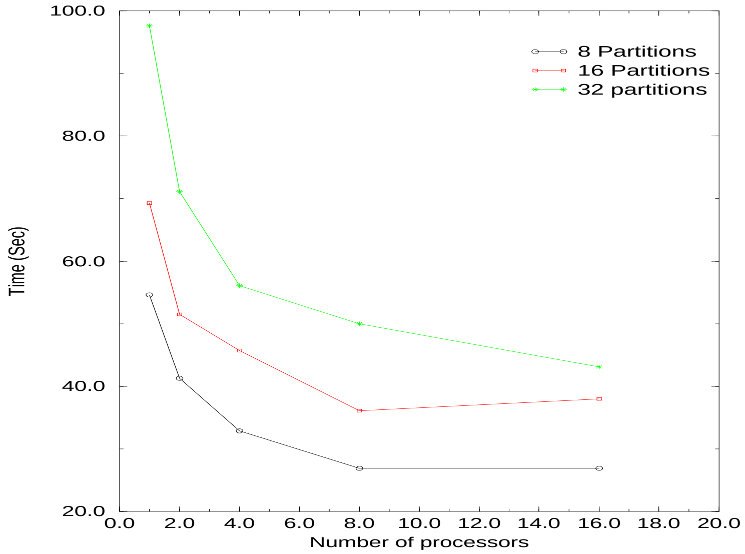


Figure 10: Preprocessing time for unstructured CFD kernel (50K Mesh)

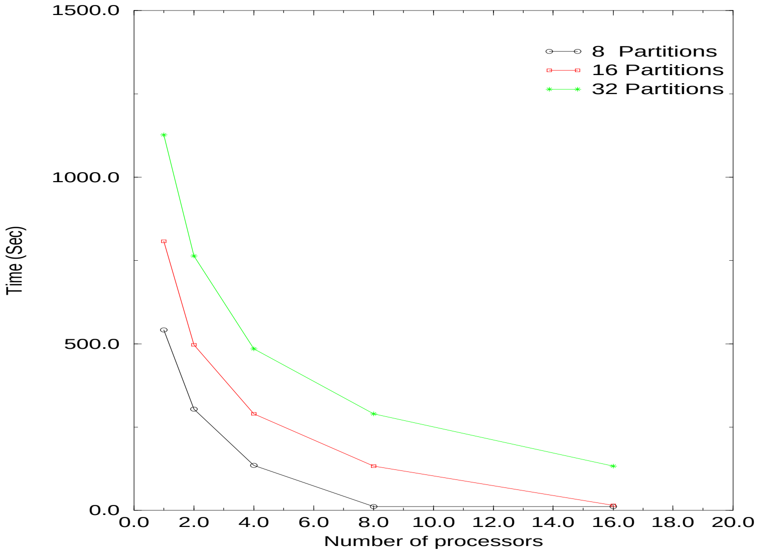


Figure 11: Computational time for unstructured CFD kernel (50K Mesh)

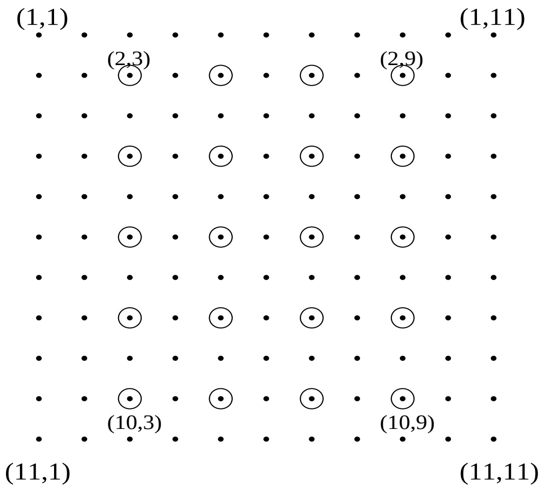


Figure 12: Accessing out-of-core array sections

array sections from disks may arise due to a number of reasons, for example FORALL or array assignment statements involving sections of out-of-core arrays.

Consider the array of size (11,11) shown in Figure 12, which is stored on disk. Suppose it is required to read the section (2:10:2,3:9:2) of this array. The elements to be read are circled in the figure. Since these elements are stored with a stride on disk, it is not possible to read them using one read call. A simple way of reading this array section is to explicitly move the file pointer to each element and read it individually, which requires as many reads as the number of elements. We call this the *Direct Read Method.* A major disadvantage of this method is the large number of I/O calls and low granularity of data transfer. Since the I/O latency is very high, this method proves to be very expensive. For example, on the Intel Touchstone Delta using 1 processor and 1 disk, it takes 16.06 ms. to read 1024 integers as one block, whereas it takes 1948 ms. to read all of them individually.

Suppose it required to read a section of a two-dimensional array specified by $(l_1 : u_1 : s_1, l_2 : u_2 : s_2)$. The number of array elements in this section is $(\lfloor (u_1 - l_1)/s_1 \rfloor + 1) \times (\lfloor (u_2 - l_2)/s_2 \rfloor + 1)$. Therefore, in the Direct Read Method,
No. of I/O requests = $(\lfloor (u_1 - l_1)/s_1 \rfloor + 1) \times (\lfloor (u_2 - l_2)/s_2 \rfloor + 1)$
No. of array elements read per access = 1
Thus in this method, the number of I/O requests is very high and the number of elements accessed per request is very low, which is undesirable.

We propose a much more efficient method called *Data Sieving* to read or write out-of-core array sections having strides in one or more dimensions. Data Sieving can be explained with the help of Figure 13. As explained earlier, each processor has an out-of-core local array (OCLA) associated with it. The OCLA is (logically) divided into slabs, each of which can fit in main memory (ICLA). The OCLA shown in the figure has four slabs. Let us assume that it is necessary to read the array section shown in Figure 13, specified by $(l_1 : u_1 : s_1, l_2 : u_2 : s_2)$, into the ICLA. Although this section spans three slabs of the OCLA, because of the stride all the data elements can fit in the ICLA.

In the Data Sieving Method, the entire block of data from column l_2 to u_2 if the storage is column major, or the entire block from row l_1 to u_1 if the storage is row major, is read into a temporary buffer in main memory using one read call. The required data is then extracted from this buffer and placed in the ICLA. Hence the name Data Sieving. A major advantage of this method is that it requires only one I/O call and the rest is data transfer within main memory. The main disadvantage is the high memory requirement. Another disadvantage is the extra amount of data that is read from disk. However, we have found that the savings in the number of I/O calls increases performance considerably. For this method, assuming column major storage,
No. of I/O requests = 1
No. of array elements read per access = $(u_2 - l_2 + 1) \times nrows$

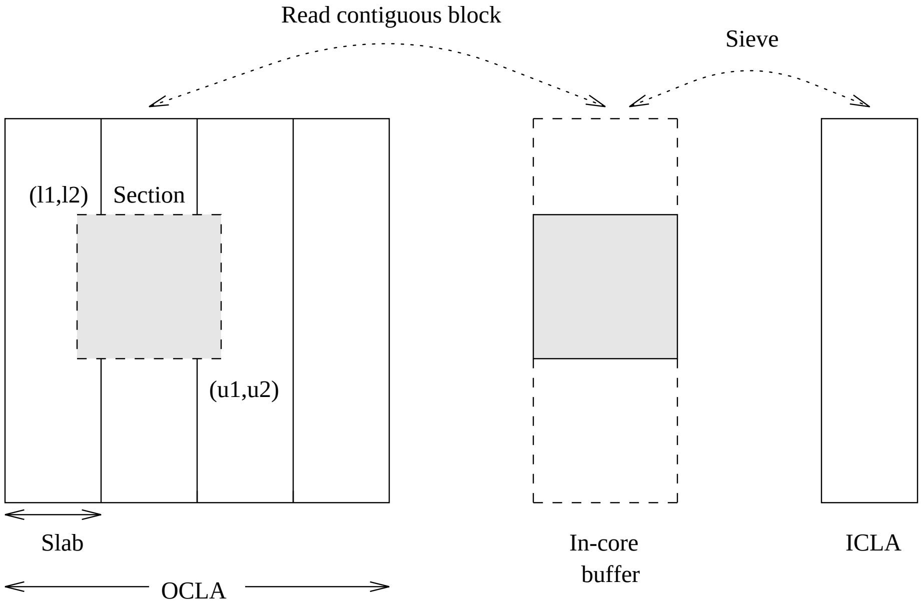


Figure 13: Data Sieving

Data Sieving is a way of combining multiple I/O requests into one request so as to reduce the effect of high I/O latency time. A similar method called *message coalescing* is used in interprocessor communication, where small messages are combined into a single large message in order to reduce the effect of communication latency. However, Data Sieving is different because instead of coalescing the required data elements together, it actually reads even unwanted data elements so that large contiguous blocks are read. The useful data is then filtered out by the runtime system in an intermediate step and passed on to the program. The unwanted data read into main memory is dynamically discarded.

Reducing the Memory Requirement: If the stride in the array section is large, the amount of memory required to read the entire block from column l_2 to u_2 will be quite large. There may not be enough main memory available to store this entire block. Since the amount of memory available to create a temporary buffer is not known, we make the assumption that there is always enough memory to create a buffer of size equal to that of the ICLA. The Data Sieving Method described above is modified as follows to take this fact into account. Instead of reading the entire block of data from column l_2 to u_2, we read only as many columns (or rows) at a time as can fit in a buffer of the same size as the ICLA. For each set of columns read, the data is sieved and passed on to the program. This reduces the memory requirements of the program considerably and increases the number of I/O requests only slightly. Let us assume that the array is stored in column major order on disk and n columns of the OCLA can fit in the ICLA. Then for this case
No. of I/O requests $= \lceil (u_2 - l_2 + 1)/n \rceil$
No. of array elements read per access $= n \times nrows$

Writing Array Sections: Suppose it is required to *write* an array section $(l_1 : u_1 : s_1, l_2 : u_2 : s_2)$ from the ICLA to the LAF. The issues involved here are similar to those described above for reading array sections. A *Direct Write Method* can be used to write each element individually, but it suffers from the same problems of large number of I/O requests and low granularity of data transfer. In order to reduce the number of I/O requests, a method similar to the Data Sieving Method described above needs to be used. If we directly use Data Sieving in the reverse direction, ie. elements from the ICLA are placed at appropriate locations in a temporary buffer with stride, and the buffer is written to disk, the data in the buffer between the strided elements will overwrite the corresponding data elements on disk. In order to maintain data consistency, it is necessary to first read the entire block from the LAF into the temporary buffer. Then, data elements from

Table 10: Performance of Direct Read/Write versus Data Sieving (time in sec.)

2K × 2K global array on 64 procs. (local array size 2K × 32), slab size = 16 columns

	`PASSION_read_section`		`PASSION_write_section`	
Array Section	Direct Read	Sieving	Direct Write	Sieving
(1:2048:2, 1:32:2)	52.95	1.970	49.96	5.114
(1:2048:4, 1:32:4)	14.03	1.925	13.71	5.033
(10:1024:3, 3:22:3)	8.070	1.352	7.551	4.825
(100:2048:6, 5:32:4)	7.881	1.606	7.293	4.756
(1024:2048:2, 1:32:3)	18.43	1.745	17.98	5.290

Table 11: I/O requirements of Direct Read and Data Sieving Methods

2K × 2K global array on 64 procs. (local array size 2K × 32), slab size = 16 columns

	No. of I/O requests		No. of array elements read	
Array Section	Direct Read	Sieving	Direct Read	Sieving
(1:2048:2, 1:32:2)	16384	2	16384	65536
(1:2048:4, 1:32:4)	4096	2	4096	65536
(10:1024:3, 3:22:3)	2373	2	2373	40960
(100:2048:6, 5:32:4)	2275	2	2275	57344
(1024:2048:2, 1:32:3)	5643	2	5643	65536

the ICLA can be stored at appropriate locations in the buffer and the entire buffer can be written back to disk.

This is similar to what happens in cache memories when there is a write miss. In that case, a whole line or block of data is fetched from main memory into the cache and then the processor writes data into the cache. This is done in hardware in the case of caches. PASSION does this in software when writing array sections using Data Sieving. Thus, writing sections requires twice the amount of I/O compared to reading sections, because for each write to disk the corresponding block has to first be fetched into memory. Therefore, for writing array sections
No. of I/O requests $= 2\lceil (u_2 - l_2 + 1)/n \rceil$
No. of array elements transferred per access $= n \times nrows$

Performance of Sieving: Table 10 gives the performance of Data Sieving versus the Direct Method for reading and writing array sections. An array of size 2K × 2K is distributed among 64 processors in one dimension along columns. We measured the time taken by the `PASSION_read_section()` and `PASSION_write_section()` routines for reading and writing sections of the out-of-core local array on each processor. We observe that Data Sieving provides tremendous improvement over the Direct Method in all cases. The reason for this is large number of I/O requests in the Direct Method, even though the total amount of data accessed is higher in Data Sieving. Table 11 gives the number of I/O requests and the total amount of data transferred for each of the array sections considered in Table 10. We observe that in the Data Sieving Method, the number of data elements transferred is more or less the same for all cases. This is because the total amount of data transferred depends only on the lower and upper bounds of the section and is independent of the stride. Hence the time taken using Data Sieving does not vary much for all the sections we have considered. However, there is a wide variation in time for the Direct Method, because only those elements belonging to the section are read. The time is lower for small sections and higher for large sections.

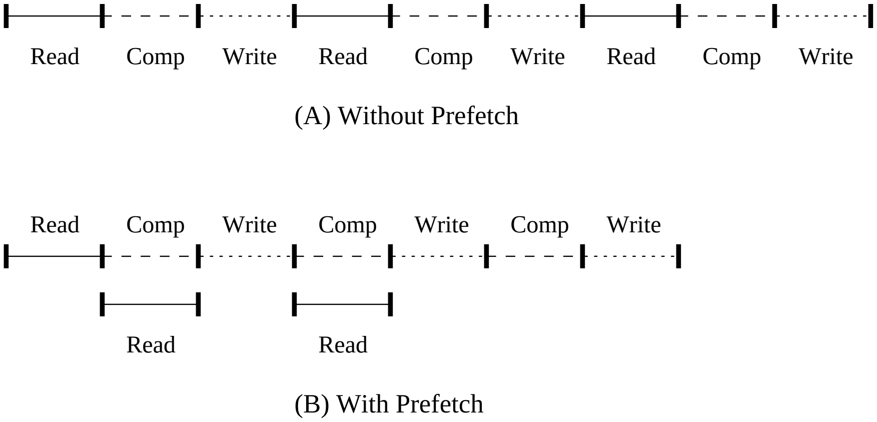


Figure 14: Data Prefetching

We observe that even for writing array sections, Data Sieving performs better than Direct Write even though it requires reading the section before writing. As expected, `PASSION_write_section()` takes about twice the time as `PASSION_read_section()` when using Data Sieving. Comparing the Direct Write and Direct Read Methods, we find that writing takes slightly less time than reading data. This is due to the way I/O is done in the Intel Touchstone Delta. The *cwrite* call returns after data is written to the cache in the I/O node, without waiting for the data to be written to disk.

All PASSION routines involving array sections use Data Sieving for greater efficiency.

4.5.2 Data Prefetching

In the Local Placement Model, the OCLA is divided into a number of *slabs*, each of which can fit in the ICLA. Program execution proceeds as follows:- a slab of data is fetched from the LAF to the ICLA; the computation is performed on this slab and the slab is written back to the LAF. This is repeated on other slabs till the end of the program. Thus I/O and computation form distinct phases in the program. A processor has to wait while each slab is being read or written as there is no overlap between computation and I/O. This is illustrated in Figure 14(A) which shows the time taken for computation and I/O on 3 slabs. For simplicity, reading, writing and computation are shown to take the same amount of time, which may not be true in certain cases.

The time taken by the program can be reduced if it is possible to overlap computation with I/O in some fashion. A simple way of achieving this is to issue an asynchronous I/O read request for the next slab immediately after the current slab has been read. This is called *Data Prefetching*. Since the read request is asynchronous, the reading of the next slab can be overlapped with the computation being performed on the current slab. If the computation time is comparable to the I/O time, this can result in significant performance improvement. Figure 14(B) shows how prefetching can reduce the time taken for the example in Figure 14(A). Since the computation time is assumed to be the same as the read time, all reads other than the first one get overlapped with computation. The total reduction in program time is equal to the time for reading two slabs, as only two of the three reads can be overlapped in this example.

Prefetching can be done using the routine `PASSION_prefetch_read()` and the routine `PASSION_prefetch_wait()` can be used to wait for the prefetch to complete. We have implemented an out-of-core Median Filtering program using Prefetching. Median Filtering is frequently used in computer vision and image processing applications to smooth the input image. Each pixel is assigned the median of the values of its neighbors within a window of a particular size, say 3×3 or 5×5 or larger. Figures 15 and 16 show the performance of Median Filtering on the Intel Touchstone Delta for windows of size 3×3 and 5×5 respectively. The image is of size 2K × 2K pixels. We observe that in all cases, prefetching improves performance significantly.

Further details about Data Prefetching are given in [TBC+94b].

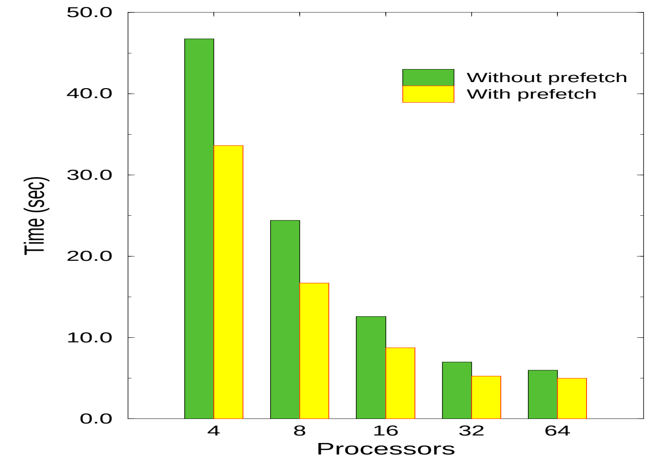


Figure 15: Median Filtering using 3 × 3 window

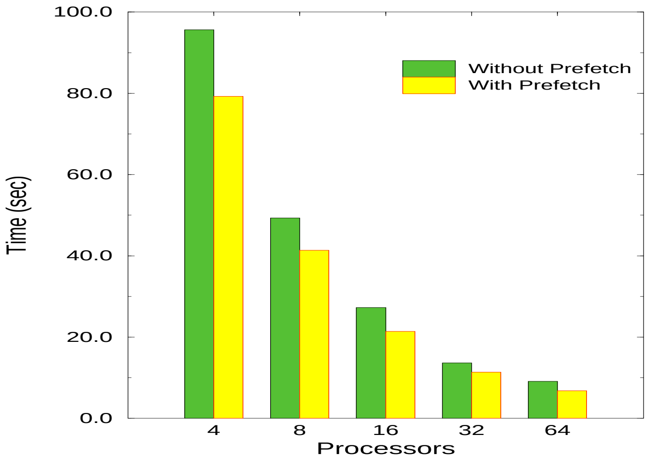


Figure 16: Median Filtering using 5 × 5 window

4.5.3 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 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. Data Reuse is described in further detail in [TBC94a].

5 Compiler Support

This section discusses the issues involved in compiling I/O intensive problems. The PASSION compiler is targeted for languages like Fortran 90D [BCF+93] and High Performance Fortran [For93], which provide explicit directives to distribute arrays across the processors of a parallel system. The compiler support is intended for programs with arrays that are too large to fit in main memory (*out-of-core* arrays) and for programs in which data has to be read from files. Such a compiler has to perform the following two main tasks:-

- Generate runtime calls to perform read/write of the arrays.
- Perform automatic program transformations to improve I/O performance.

A number of compiler techniques have been developed for in-core programs. Similar techniques can be used for compiling out-of-core programs. This section briefly describes some of the important steps in compiling out-of-core programs and some compiler techniques used in the PASSION compiler.

The PASSION compiler compiles an out-of-core (OOC) HPF program in two phases. The first phase performs preprocessing of the source HPF program in the global name space. In the second phase, optimizations on the corresponding node program are carried out. Figure 17 shows the phases in compiling an OOC program using the PASSION compiler. While the first phase is independent of the underlying execution model, optimizations in the second phase differ according to the execution models.

- Phase I: *Global Program Preprocessing*

  In the first phase, dataflow analysis is performed to obtain flow information about the scalar and array variables.

  - *Global Dataflow Analysis and* `FORALL` *Optimizations:* Preprocessing of the SPMD program involves performing analysis of the source program in the global name space using standard dataflow techniques. The main aim of the dataflow analysis is to examine the access patterns of the program variables. We are interested in finding (1) when a particular program variable is first accessed (read/written), (2) how many times this variable is defined (written), (3) where is the last access of this variable, (4) how many elements of an array are used in more than one access, (5) which array dimension is accessed more often. This information helps the compiler to compute the *range* of each array variable. In addition, the compiler can check whether the array statements could be reorganized so that the statements that access the same program variables are closer in the program space. Dataflow analysis also provides dependence information about the `FORALL` and array assignment statements. Using interval analysis, redundant computation can also be detected. Using this information, multiple `FORALL` statements can be merged to remove redundant computation.

- Phase II: *Local Program Optimizations*

  In the second phase, the PASSION compiler operates in the local name space. The second phase involves four parts, (1) Work Distribution, (2) Loop Optimizations, (3) Local Dataflow Analysis, (4) Communication and I/O Optimizations and (5) Inter and Intra File Reorganization.

  - *Work Distribution:* To distribute work among processors, the compiler uses distribution information provided by the compiler directives. Work distribution involves (1) computing local array sizes for each processor, (2) scalarizing the `FORALL` statement and generating corresponding `DO` loops in the local name space. All necessary communication is detected and the corresponding communication sets are computed.
  - *Loop Optimizations:*
    - *Memory Transformations:* In out-of-core programs, computation is performed on data which is present in the primary memory of compute nodes. This requires fetching *slabs* of data from disk and computing on the in-core slabs. As a result, computation has to be reordered. This

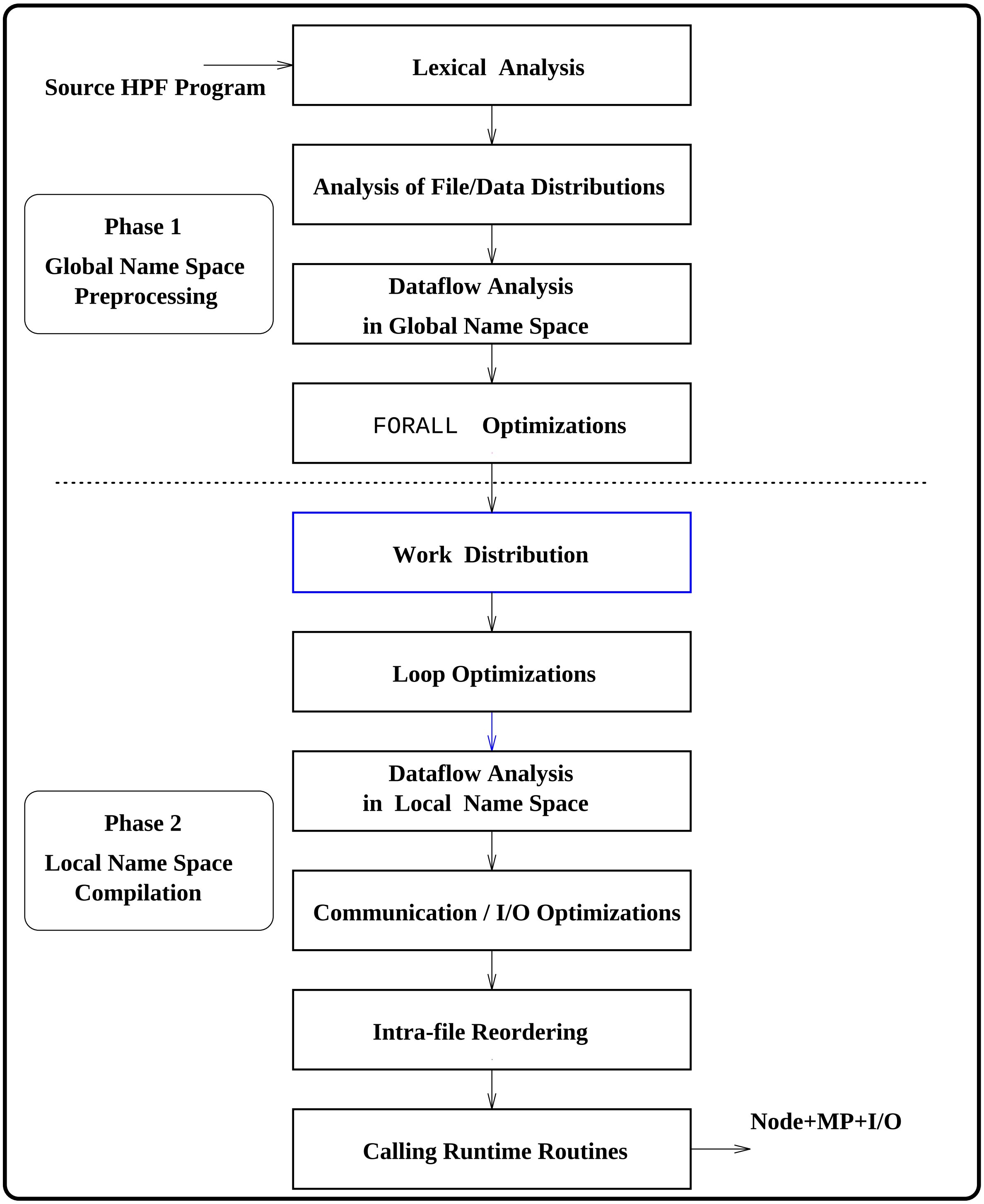

Source HPF Program
Lexical Analysis
Analysis of File/Data Distributions
Phase 1
Global Name Space Preprocessing
Dataflow Analysis in Global Name Space
FORALL Optimizations
Work Distribution
Loop Optimizations
Dataflow Analysis in Local Name Space
Phase 2
Local Name Space Compilation
Communication / I/O Optimizations
Intra-file Reordering
Calling Runtime Routines
Node+MP+I/O

Figure 17: PASSION Compiler Phases

reordering can be done by stripmining the computation. The local computation corresponding to a `FORALL` statement is stripmined according to the available memory. Depending on the underlying execution model, different strategies are used to stripmine local computation. The information about the amount of available memory is obtained either from compiler directives or at runtime.

    * *Locality Transformations:* The time required to access data from disks depends on how data is accessed and how data is stored on disks. In order to reduce the I/O cost, either the computation can be reordered (iteration blocking, tiling [Iri88, SD90, WL91, CK89]) according to the placement of data in files (to take advantage of data locality) or data can be reordered in files according to the computation. Reordering of data in files results in extra overhead. However, computation reordering may not always be trivial. In such cases, data reorganization on files is preferred. Locality analysis also helps a compiler to predict when to prefetch data. Data prefetching can be used to hide memory latency and disk accesses can be overlapped with computation.

- *Local Dataflow Analysis:* In this step, the local program is analyzed for dependencies of program variables in the local name space. The dataflow information provides information about where a particular program variable is accessed. Also, dataflow analysis provides information about which program variables are updated and which processors require the updated variables. This information is used for optimizing communication and I/O.

- *Communication Optimizations:* In out-of-core compilation, communication can be performed using two different methods, *In-core Communication* and *Out-of-core Communication*. In the In-core Communication Method, the data corresponding to each slab is fetched when the computation on the slab is performed. In the Out-of-core Communication Method, data required by the entire out-of-core array is fetched. The choice between the two methods depends on the type of computation inherent in the program. Information provided by the local dataflow analysis can be used to aggregate communication and to place communication calls in suitable places.

- *I/O Optimizations:* The I/O cost associated with a program can be measured either using the number of I/O requests or using the number of disk requests. There are several ways of optimizing the number of I/O (disk) requests. From our previous work, it has been observed that several I/O requests with small request sizes degrade the I/O performance. Hence to achieve high performance, multiple I/O requests can be aggregated. Using local dataflow analysis, the compiler determines what data can be accessed using a single I/O request. The dataflow information can also be used to place I/O calls so that the overall I/O cost can be reduced. Strategies like two-phase access and disk-directed I/O [Kot94] can be used to optimize I/O from disks.

- *Inter and Intra File Organizations:* The final step in the local program optimization involves organization of data across files and within files. Depending on the underlying execution model, the compiler generates local array files. The data is either reorganized into the required format or written into local files. The second optimization involves reorganizing data within each array file. Both global and local dataflow analyses provide information about variable access patterns. Using this information, the data within a file can be arranged so as to match the access pattern.

5.1 Language Support

For efficient compilation of out-of-core programs, the compiler requires information about the distribution of arrays on processors as well as on the disks. Languages like Fortran D, High Performance Fortran provide explicit directives to describe distribution of data on processors. However, no directives are provided to specify data distribution on disks.

Various attempts have been made to provide compiler directives for describing data distribution on disks (Vienna Fortran [BGMZ92, ZBC$^+$92], HPF [Sni92]). However at present there is no consensus about how the distribution information should be passed to the compiler. There exist several problems in defining such directives. The out-of-core arrays used in the programs are stored as files. Hence in case of out-of-core problems, two distinct distributions exist. First is the array distribution and second is the file distribution. Most of the high performance file systems distribute files over disks. Distribution of the file depends on many factors such as number of disks, type of data striping and the language used for computation. Files can be created during the programs or they may be already stored on the file system. As a result, the file size may not be known at compile time. Also, one or more arrays (which can have different distributions) can be initialized from the same file. Compiler directives should be designed to provide all the above information. Since the design of the file system is architecture dependent, these directives should provide a portable interface to the underlying file system.

In the case of the PASSION compiler, compilation depends on the programming model as well as on the underlying execution model (Local Placement Model or Global Placement Model). The user can direct the compiler to choose a particular execution model. In addition, the PASSION compiler provides a set of compiler directives to provide information about distribution of arrays and files. Some of the proposed directives are described below.

• **DISKS**: This directive is used for describing the logical mapping of disks over which one or more files may be distributed and/or which are used to distribute scratch files for out-of-core computations. The syntax for this directive is similar to the **PROCESSORS** directive in HPF. For example,

DISKS D(8,8)

indicates that disks are logically arranged as a two-dimensional logical grid of size 8×8. This directive enables a compiler to associate a disk (or a set of disks) with processors for file distributions and out-of-core computations. Many processors are allowed to be associated with one disk and many disks are allowed to be associated with one processor.

• **FILEPROC**: This directive is also similar to the **PROCESSORS** directive in HPF except that it specifies the processors which really participate in performing I/O. From our earlier studies [BdRC93, Bor93], we observed that the best performance need not necessarily be obtained when all processors performing computations also perform I/O. Thus, this provides the user the flexibility to specify a set of processors to perform I/O. This directive is optional, and if not specified, the default is the number of processors specified in the **PROCESSORS** directive. For example,

FILEPROC FP(2,2)

specifies that a 2×2 array of processors participates in I/O.

• **FILEDISTR**: This directive declares a file-template and distributes it over the specified number of disks declared in the **DISK** directive. It also uses the optional **FILEPROC** parameter. This directive uses names declared in **DISKS** and **FILEPROC** as pointers to the corresponding topologies. For example,

FILEDISTR F(D,[FP])

declares a file-template F which is distributed over D disks, and it associates this template with the processors declared in FP. Thus a file distributed over a set of disks can be associated with different sets of processors by using this directive. For example, when declared together,

FILEDISTR F(D, FP1)
FILEDISTR F(D, FP2)

permit two different processor configurations to access files on the same set of disks.

• **FILEALIGN**: This directive is similar to the **ALIGN** directive of HPF. **FILEALIGN** aligns the list of associated files to the template declared using **FILEDISTR** directive. However, there is a fundamental difference between **ALIGN** and **FILEALIGN**. A file may not have a size at declaration time. Thus the same file may be aligned to more than one file-templates as illustrated above. This is quite logical since a file can be opened by two different processor grids. Following example illustrates the **FILEALIGN** directive.

FILEALIGN F :: F1, F2, F3

• **ASSOCIATE**: This directive describes the relationship between an array's and the corresponding file's mapping. That is, the **ASSOCIATE** directive associates a file-template with the corresponding array template. **ASSOCIATE** directive has the following form

ASSOCIATE :: (file-template, array-template)

For example,

ASSOCIATE :: (F,A),(,),...

associates the file-template F with the array-template A. Thus, this directive provides an HPF compiler a list of files to be used for I/O for a set of arrays aligned to the corresponding array template.

- **OUT_OF_CORE**: This directive declares an array as an out-of-core array. The following example declares array A as an out-of-core array.

OUT_OF_CORE :: C

6 File System Support

PASSION interacts mainly with the file system handler portion of the underlying operating system. This interaction includes providing information about data access patterns for parallel I/O and sending actual data access requests. The information can originate either from the PASSION runtime calls embedded in the application or from the two-phase manager. This information can be broadly classified into following categories:

- Hints for prefetching data.
- Data distribution and redistribution on disks (GPM to LPM, LPM to GPM, etc.).
- File system parameters (stripe size, number of disks for striping etc.)

The file system processes the information provided to it to generate a schedule for disk I/O and to determine data file layout on disks. It also passes feedback information (load, prefetch and current queue sizes etc.) to the Two-Phase Manager and the Runtime System.

The I/O subsystem in PASSION assigns priorities to the processing of access requests and different types of information provided by other layers. Current access requests are serviced immediately because access requests result in blocking of processes until the request is completed. Similarly, information about data distribution and redistribution is processed immediately. Prefetching (on the basis of hints provided) on the other hand is done in disk idle time when there is no pending disk access request or higher priority information to be processed (Figure 18, lines 13–17). Whenever there are no pending disk requests, the prefetch information is analyzed to produce a schedule of attached disks for all active processes performing I/O. Scheduling is done so as to give representation to all active processes in the computational array and to keep the prefetch cache full at all the I/O nodes.

The PASSION I/O subsystem ensures that prefetch information is updated after each read request. Updation consists of releasing the cache block in the case of a cache hit or removing prefetch entry in the case of a cache miss. This ensures that data which will not be required in the future does not occupy the cache or if a particular process has a higher I/O request rate, it does not overrun the cache. A vacancy in the cache triggers more prefetching when the I/O node is idle.

Changes in access patterns of processes during run time are also accommodated. Thus if a process has a conditional I/O access, it may still inform about it. The file system prefetches that information if it has sufficient cache space and time. And finally, when the accesses are actually made, it deletes all the requests which have become obsolete. For this, it relies on the ordering of requests which is ensured by the suggested specification scheme. Thus, for all read requests from the computational node to the I/O node there are following possibilities :

- **Hit** in the prefetch cache : Data requested is returned and the prefetch manager is informed. The prefetch manager deletes all requests that will not be possible any more (like those in other conditional branch) and schedules more reads for the space now available in the cache.

```
PASSION I/O SERVER
BEGIN
    DO FOREVER
        IF disk access request (READ/WRITE) for data D THEN
            IF D not prefetched THEN
                Record prefetch Miss
                Schedule an immediate request for D on disk
            ELSE IF D prefetched THEN
                Record prefetch Hit
                Deallocate resources held for D.
                Deallocate resources with conditional data
            END IF
        ELSE IF prefetch hints THEN
            Save them
        ELSE IF data dist. or file parameter info. THEN
            set system parameters
        ELSE IF Idle THEN
            CALL Prefetch_Scheduler
        END IF
    END DO
END
```

Figure 18: PASSION I/O server process

- **Miss** in the prefetch cache : There are two situations. One is that there were no prefetch requests associated with this data item. In that case it informs the prefetch manager to update the state (access rates etc.) of this process. The second case is when there was a prefetch request associated with it but it could not be completed either because the rate of access of this process is greater than what the prefetch manager predicted, or there was not enough cache available to service it. In this case, the prefetch manager schedules this request immediately on the disk and updates its entries as in the case of a hit.

In addition to this, it also provides for locality based caching as a default caching scheme (whenever there is enough cache left after servicing prefetch requests).

6.1 Performance Results for prefetching

Table 12: Performance results for random file accesses from a shared file in 8-Kbyte blocks on Intel iPSC/860

No. of Accesses per processor	No. of Processors (P)	Without Prefetching Time(ms)	With Prefetching Time(ms)
50	1	789	277
50	2	870	294
50	4	1780	296
100	1	1461	555
100	2	1601	584
100	4	4055	640

We used Intel iPSC/860 (16 processor nodes, 2 I/O nodes) for the experimental evaluation of the prefetching technique. Modifications were made in the Intel Concurrent File System (CFS) to incorporate all the features described earlier. Table 12 shows the results for random accesses to a shared file using Mode 0 of

Table 13: Performance of out-of-core matrix multiplication

Matrix Size	No. of Processors	Without Prefetching	With Prefetching
N×N	(P)	Time(Sec)	Time(Sec)
512	1	581.5	570.1
512	4	233.7	194.9
512	8	176.3	159.6

the CFS (each node maintains its own file pointer and can access information anywhere in the file). The accesses had a fairly equal distribution over both the I/O nodes. It shows performance improvement of up to 80% over the case without prefetching. With the increase in the number of processors from two to four, the performance without prefetching was poor primarily because I/O nodes have to service twice as many requests. With prefetching on the other hand, data blocks had already been prefetched when the request arrived. We also tested an out-of-core matrix multiplication algorithm discussed in [BCT94] and generated access pattern information using read dependency analysis. The results obtained are listed in Table 13. It can be observed that prefetching improves performance considerably.

7 VIP-FS: A *VI*rtual *P*arallel *F*ile *S*ystem

As parallel I/O techniques become more intricate and complex, they become more cumbersome for users to take advantage of. In an effort to provide a simple straight-forward interface to parallel I/O that is available in many environments we have developed VIP-FS, a *VI*rtual *P*arallel *F*ile *S*ystem [dRHC94].

VIP-FS is a portable parallel file system for distributed computing that provides high level data mapping abstractions. The file system is deemed a *virtual* file system because it is implemented using multiple individual standard file systems integrated by a message passing system. VIP-FS is portable across many architectures as well as many message passing systems and is designed to work in a heterogeneous environment.

7.1 Functional Description

VIP-FS has two functional layers: the *P*arallel *F*ile *I*nterface layer (PFI) and the *L*ocal *D*evice *I*nterface layer (LDI). Figure 19 illustrates the logical configuration of VIP-FS.

7.1.1 PFI: The Parallel File Interface

User applications interact with VIP-FS through the PFI layer. A user defined data mapping is passed to the PFI layer for all files opened or created. This data mapping gives each user application process a ”local” view of some subset of the global data. Data access requests are then passed to the PFI layer where the mapping functions determine where the data should reside. From this, the PFI layer distributes/gathers the data according to the mapping function to/from the appropriate physical I/O devices through the LDI layer.

7.1.2 LDI: The Local Device Interface

The LDI layer is for the most part a stream-lined request filler. As requests from the PFI layer are received, the data is immediately read/written to the specified location and the result returned. No computation or synchronization is done on the LDI side of the system. The only exception to this is in the case of collective read accesses.

Collective read access is a modified form of the two-phase access paradigm developed by del Rosario, Bordawekar, and Choudhary [dRBC93]. The collective read access is initiated by each application process making a request for the same set of data from the global data view (the data requested can and generally will contain regions that are outside each of the computational process' ”local” data view as specified by the data mapping). Exactly one PFI layer process transmits the request to all LDI processes. The LDI can then

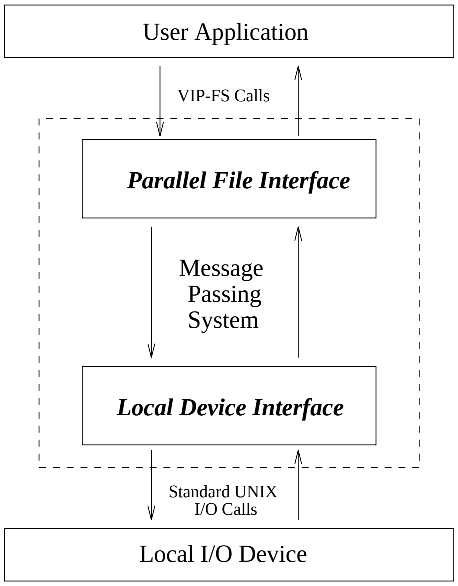


Figure 19: VIP-FS Logical Overview

determine what portion of the data request resides locally and access the data in the most efficient manner. The LDI then parses the data accessed and delivers the data segments to the appropriate computational processes as determined by the user specified data mapping.

By allowing the collective accessing of data as described above we gain performance benefits in three areas:

- Fewer read requests are sent across the network.
- Data can be accessed efficiently according to its physical distribution, rather than accessing it according to individual data requests.
- Data is delivered directly to the proper computational processes as specified by the data mapping. No extra swapping of data is necessary among computational processes.

However, this type of access requires the LDI layer to have knowledge of the data mapping and to calculate the mapping function for the above described collective read accesses, thus adding a level of complexity to the LDI layer that would otherwise not be present.

7.2 I/O Subsystem

A key feature of VIP-FS is versatility. Instead of mandating a particular I/O subsystem with fixed I/O nodes and fixed compute nodes, we allow the user to configure the system in any fashion desired.

The only knowledge the user needs of the underlying I/O system is which nodes have local I/O devices. Any combination of I/O nodes and computational processes may be distributed in any fashion desired with the only restriction being that there may be at most one I/O process per node. Figure 20 shows distribution possibilities of computational processes and I/O nodes.

There is also the option to the user of not specifying any I/O node setup in which case a system specific default will be used to identify available I/O nodes. Thus, the user needs no knowledge of the underlying I/O subsystem to take advantage of the parallel I/O facilities provided by VIP-FS. However, if the application would benefit from a customized I/O setup, the user has the ability to configure the system to the application's specific needs.

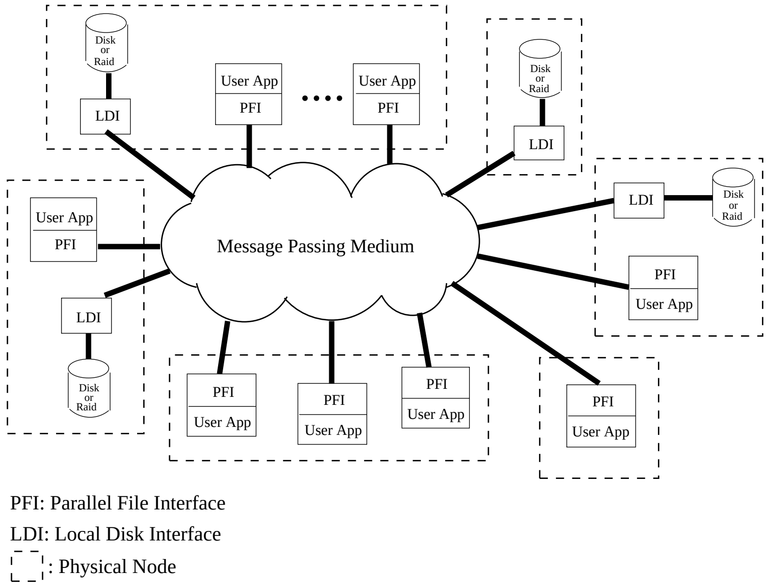


Figure 20: VIP-FS Distribution over Physical Nodes

8 Parallel I/O for Integrating Task and Data Parallelism

I/O techniques have been used in the past in communicating data among sequential tasks. A Unix pipe is an example of such an approach. This component of PASSION involves developing parallel pipes to facilitate communication between data parallel tasks (potentially executing on different architectures). Given that the distribution of data, the number of processors etc. may be different in the two communicating data parallel tasks, several issues need to be addressed. These include redistribution of data at the time of communication, protocol for communication, providing information about the distribution in one task to the other etc. This work addresses these issues.

In an integrated data/task parallel system [FAXC94], CHANNELS can be used as a mode of communication between data parallel tasks. We use parallel I/O techniques to implement CHANNELS. A CHANNEL provides a uniform mode of communicating data between two data parallel tasks. Programs are constructed by using CHANNELS to plug together concurrent tasks. This provides a many-to-many communication model between different processes of the communicating tasks. Figure 21 shows two tasks $T1$ and $T2$ connected by a CHANNEL C. The two tasks are assumed to be data-parallel executing on m and n processors respectively.

The features of CHANNELS include the following:-

- *Distribution Independence:* Tasks on the two ends of a CHANNEL may have different data distributions. A CHANNEL provides a uniform mode of communicating data which is distribution independent. For example, data in task $T1$ may be distributed in a block fashion and data in task $T2$ may be distributed in a cyclic fashion.

- *Information Hiding:* A CHANNEL provides an interface between two communicating tasks to facilitate information hiding. Task $T1$ need not be aware of the data distribution in task $T2$ and vice versa. It pushes the data for communication to one end of the CHANNEL, and also specifies its data distribution. The other task can request the data from the CHANNEL in its own distribution format.

- *Synchronization:* The communicating tasks use the CHANNEL for synchronization. The receiving task has to wait for the CHANNEL to get full before it can proceed.

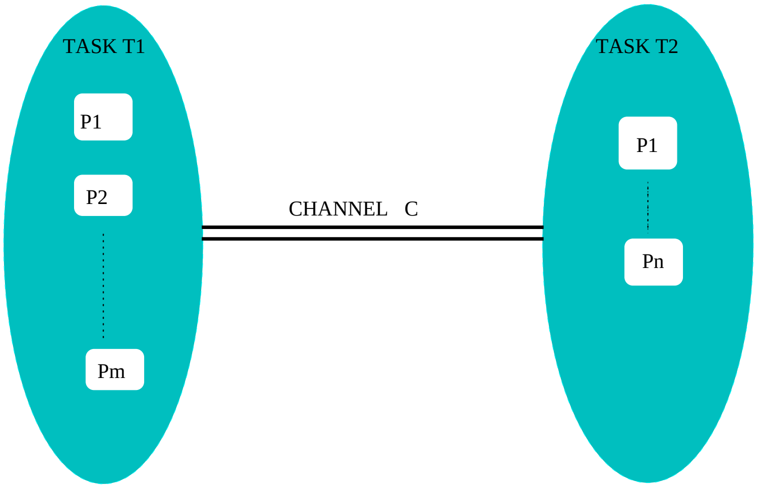


Figure 21: CHANNEL connection between two tasks

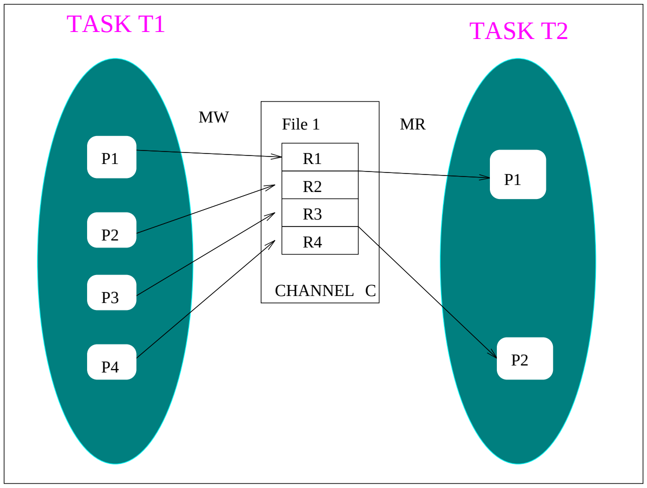


Figure 22: Shared File Model

8.1 Shared File Model - SFM

In this model, a CHANNEL is implemented using a shared file as illustrated in Figure 22.

$T1$ and $T2$ are two tasks connected by a CHANNEL C. The CHANNEL is unidirectional with $T1$ as the sending task and $T2$ as the receiving task. Data D is communicated over the CHANNEL. The shared file consists of a number of regions R_i. Process P_i of task $T1$ writes to region R_i of the shared file. This region may be contiguous or striped.

The mapping function MW provides a one-to-one mapping between the processes P_i executing task $T1$ and the regions R_i of the shared file. The mapping function MR maps the regions R_i of the shared file to the processes P_i executing task $T2$. MR can be a one-to-many mapping as we allow many processes to read from the same file region even though only one process can write to a given region. Similarly MR can also be a many-to-one mapping.

Multiple-Readers/Multiple-Writers are allowed in this model for implementing CHANNELS. The mapping function MW could map a process either to a contiguous region or a striped region on the file. The information about the mapping is specified at the beginning of the file. The mapping function MR is defined by using this information at the top of the file. *Synchronization* is achieved through a synchronization variable at the beginning of the file. *Ports* for a given CHANNEL are defined by the file pointers opened at each end of the CHANNEL. The files need not be stored on disks. They are stored in memory buffers and when the buffers overflow, they are transferred to disks. The files are reclaimed once they are read. This approach allows dissimilar sets of processors to communicate as long as the file formats are the same. Hence this approach extends easily to a heterogeneous environment.

8.2 Multiple File Model (MFM)

One of the limitations of the SFM is that it does not allow the data to be communicated in a pipelined fashion. If a large data structure has to be transferred from one task to another, the transfer can be pipelined by breaking the data structure into a number of smaller data sets. Each set has a synchronization variable associated with it. Multiple files can provide this abstraction of communication. This model implements a CHANNEL using a multiple file system. This is similar to the previous model except that it offers multiple files as an intermediate storage facility. In the MFM, the mapping function MW gives a file name as well as a file region to map the data of a process in a task. A process can typically have more than one file on which its data is mapped. Synchronization variables are associated with each file or a set of files in the MFM. Since there are multiple synchronization variables, the communication over the CHANNEL can be pipelined. The files in this case are smaller as the data is distributed over multiple files.

We have implemented these two models for an image processing application on the Intel IPSC-860, details about which can be found in [ACFK94].

9 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] which supports logical partitioning of files. PIOUS [MS94] is a parallel file system for a networked computing environment. 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]. There are several schemes that allow for the exploitation of access pattern information. Crockett [Cro89] discusses parallel file accesses in relation to possible storage techniques. Kotz et. al. [EK89] use pattern predictors to predict an application's future access patterns to perform prefetching. More recently, Patterson et. al. [PGS93] discuss the benefits of disclosing application level hints about future I/O accesses. Prefetching for in-core problems is discussed in [MLG92, CKP91]. The effects of prefetching blocks of a file in a multiprocessor file system are studied in [D. 90].

Joel Saltz and his group at the University of Maryland have developed the PARTI/CHAOS toolkit, which is a collection of runtime library routines to handle in-core irregular computations [DSB91, SBW91]. Compilation methods for irregular problems have been investigated by Ponnusamy [Pon94], Das [DPSM92] and Hanxleden [vKK+92].

There has been a lot of work done in compiler optimizations for data locality. These techniques are also applicable for compiling out-of-core programs. Abu-Sufah investigates strategies to improve performance of fortran programs in virtual memory environment [AS79]. Compiler transformations such as tiling, strip-mining, loop interchange, loop skewing are proposed by Wolfe [Wol89b, WB87, Wol87, Wol89a]. Transformations like Unroll-and-Jam and Scalar Replacement are proposed by Carr [Car93, CK94]. Callahan studies the problem of register allocation [CCK90]. Irigoin and Triolet also propose transformations to improve locality [Iri88]. An excellent description of compiler transformations is given in [BGS93]. Wolf and Lam propose an elegant loop transformation theory to improve locality and parallelism [WL91, Wol92]

Language extensions for out-of-core data parallel programs are proposed by the Vienna Fortran group [BGMZ92]. Marc Snir of IBM has submitted a proposal [Sni92] for I/O in HPF to the HPF Forum.

10 Conclusions

PASSION provides software support for high performance parallel I/O on distributed memory parallel computers. It provides support for compiling out-of-core data parallel programs, parallel input-output of data and parallel access to files, communication of out-of-core data, redistribution of data stored on disks, many optimizations including Data Prefetching from disks, Data Sieving, Data Reuse etc., as well as support at the file system level. PASSION also provides an initial framework for runtime support for out-of-core irregular problems. This report gives a brief overview of the various components of PASSION, together with some performance results on real applications.

All the runtime procedures, optimizations and file system support described in this report have been implemented. A subset of the compiler has been implemented and a full implementation is in progress. PASSION is currently available on the Intel Paragon, Touchstone Delta and iPSC/860 using Intel's Concurrent File System. Efforts are underway to port it to the IBM SP-1 and SP-2 using the Vesta Parallel File System.

11 PASSION Related Papers

Additional information about PASSION is available on the World Wide Web at `http://www.cat.syr.edu/passion.html`. PASSION related papers can also be obtained from the anonymous ftp site `erc.cat.syr.edu` directory `ece/choudhary/PASSION`, or from `ftp.npac.syr.edu` directory `users/choudhar`. The following is the list of papers related to the PASSION project and their corresponding file names:-

- `ics94-out-of-core-hpf.ps.Z`: "Compiler and Runtime Support for Out-of-Core HPF Programs", Rajeev Thakur, Rajesh Bordawekar and Alok Choudhary, *Proc. of Int. Conf. on Supercomputing (ICS 94)*, July 1994, pp. 382-391.

- `splc94_passion_runtime.ps.Z`: "PASSION Runtime Library for Parallel I/O", Rajeev Thakur, Rajesh Bordawekar, Alok Choudhary, Ravi Ponnusamy and Tarvinder Singh, *Proc. of the Scalable Parallel Libraries Conference*, Oct. 1994

- `passion_report.ps.Z`: "PASSION: Parallel and Scalable Software for Input-Output", Alok Choudhary, Rajesh Bordawekar, Michael Harry, Rakesh Krishnaiyer, Ravi Ponnusamy, Tarvinder Singh and Rajeev Thakur, *NPAC Technical Report SCCS–636*, Sept. 1994.

- `access_reorg.ps.Z`: "Data Access Reorganizations in Compiling Out-of-core Data Parallel Programs on Distributed Memory Machines ", Rajesh Bordawekar, Alok Choudhary and Rajeev Thakur, *NPAC Technical Report SCCS–622*, Sept. 1994.

- `vipfs.ps.Z`: "The Design of VIP-FS: A Virtual Parallel File System for High Performance Parallel and Distributed Computing", Juan Miguel del Rosario, Michael Harry and Alok Choudhary, *NPAC Technical Report SCCS–628*, May 1994.

- `adopt.ps.Z`: "ADOPT: A Dynamic Scheme for Optimal Prefetching in Parallel File Systems", Tarvinder Singh and Alok Choudhary, *NPAC Technical Report SCCS–627*, 1994.

- `task_data.ps.Z`: "Integrating Task and Data Parallelism Using Parallel I/O Techniques", Bhaven Avalani, Alok Choudhary, Ian Foster and Rakesh Krishnaiyer, *Proc. of the Int. Workshop on Parallel Processing*, Bangalore, India, Dec. 1994.

Acknowledgments

This work has been influenced by a number of people. We are thankful to Ken Kennedy, Chuck Koelbel, Geoffrey Fox, Joel Saltz and Paul Messina for collaboration on various parts of this work and many enlightening discussions. We thank Justin Rattner and Dave Riss of Intel SSD for recognizing the importance and providing support for this work. Brad Rullman of Intel SSD has provided technical input on various aspects. We thank Marc Snir of IBM Corp. for his encouragement, collaboration, technical insight and support for this work. We thank Yarsun Hsu and the entire parallel I/O group at IBM Yorktown Heights for many fruitful discussions. We thank Rick Stevens, Ian Foster, Mani Chandy and Dan Reed for collaboration and technical discussions. Terry Pratt has provided technical input and support through CESDIS. Robert Ferraro of JPL has provided many insights into applications needing massively parallel I/O solutions.

Mike del Rosario has collaborated with us in the implementation of VIP-FS and the Two Phase Data Access Method. We thank Ravikumar Muppirala of the Chemistry department at Syracuse University for providing us the protein molecule structures used in this paper.

References

[ACFK94] B. Avalani, A. Choudhary, I. Foster, and R. Krishnaiyer. Integrating Task and Data Parallelism Using Parallel I/O Techniques. In *to appear in Proceedings of the International Workshop on Parallel Processing, Bangalore, India*, December 1994.

[AS79] W. Abu-Sufah. *Improving the Performance of Virtual Memory Computers.* PhD thesis, Dept. of Computer Science, University of Illinois, 1979.

[BBO$^+$83] B. Brooks, R. Bruccoleri, B. Olafson, D. States, S. Swaminathan, and M. Karplus. CHARMM: A Program for Macromolecular Energy Minimization and Dynamic Calculations. *Journal of Computational Chemistry*, 4:187, 1983.

[BCF$^+$93] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka. Fortran 90D/HPF compiler for distributed memory MIMD computers: Design, implementation, and performance results. In *Proceedings of Supercomputing '93*, pages 351–360, November 1993.

[BCT94] R. Bordawekar, A. Choudhary, and R. Thakur. Data Access Reorganizations in Compiling Out-of-core Data Parallel Programs on Distributed Memory Machines. Technical Report SCCS–622, NPAC, Syracuse University, April 1994.

[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.

[BGS93] D. Bacon, S. Graham, and O. Sharp. Compiler Transformations for High-Performance Computing. Technical Report UCB/CSD-93-781, Computer Science Division, University of California, Berkeley, Computer Science Division, University of California, Berkeleyley, California 94720, 1993.

[Bor93] R. Bordawekar. Issues in Software Support for Parallel I/O. Master's thesis, Dept. of Electrical and Computer Engineering, Syracuse University, May 1993.

[Car93] Steve Carr. *Memory-Hierarchy Management.* PhD thesis, Rice University, February 1993.

[CCK90] David Callahan, Steve Carr, and Ken Kennedy. Improving Register Allocation for Subscripted Variables. *Proc. of SIGPLAN'90 Conference on Program Language Design and Implementation*, June 1990.

[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.

[CK89] S. Carr and K. Kennedy. Blocking Linear Algebra Codes for Memory Hierarchies. *Proc. of the Fourth SIAM Conference on Parallel Processing for Scientific Computing*, 1989.

[CK94] Steve Carr and Ken Kennedy. Scalar Replacement in the Presence of Conditional Control Flow. *Software-Practice and Experience*, 24(1):51–77, January 1994.

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

[Cro89] T. Crockett. File Concepts for Parallel I/O. In *Proceedings of Supercomputing '89*, pages 574–579, 1989.

[D. 90] D. Kotz and C. 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.

[DPSM92] R. Das, R. Ponnusamy, J. Saltz, and D. Mavriplis. Distributed Memory Compiler Methods for Irregular Problems – Data Copy Reuse and Runtime Partitioning. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers and Runtime Environments for Distributed Memory Machines*, pages 185–220. Elsevier Science Publishers, 1992.

[dRBC93] J. del Rosario, R. Bordawekar, and A. Choudhary. Improved parallel i/o via a two-phase runtime access strategy. In *Proceedings of the Workshop on I/O in Parallel Computer Systems at IPPS '93*, April 1993.

[dRC94] J. del Rosario and A. Choudhary. High performance i/o for parallel computers: Problems and prospects. *IEEE Computer*, March 1994.

[dRHC94] J. del Rosario, M. Harry, and A. Choudhary. The Design of VIP-FS: A Virtual Parallel File System for High Performance Parallel and Distributed Computing. Technical Report SCCS-628, NPAC, Syracuse University, May 1994.

[DSB91] R. Das, J. Saltz, and H. Berryman. A Manual for PARTI Runtime Primitives. Interim Report 17, ICASE, NASA Langley Research Center, May 1991.

[DSE88] P. Dibble, M. Scott, and C. 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.

[EK89] C. Ellis and D. Kotz. Prefetching in file systems for MIMD multiprocessors. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages I:306–314, August 1989.

[FAXC94] I. Foster, B. Avalani, M. Xu, and A. Choudhary. A Compilation System that Integrates High Performance Fortran and Fortran M. In *Proceedings of the Scalable High Performance Computing Conference*, May 1994.

[For93] High Performance Fortran Forum. High Performance Fortran Language Specification Version 1.0. Technical Report CRPC-TR92225, Center for Research in Parallel Computing,Rice University, January 1993.

[FWM94] G. Fox, R. Williams, and P. Messina. *Parallel Computing Works*. Morgan Kaufmann Publishers Inc., 1994.

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

[Iri88] Francois Irigoin. Code Generation for the Hyperplane Method and for Loop Interchange. Technical Report E102, Ecole Des Mines De Paris, October 1988.

[Kot94] D. Kotz. Disk-directed I/O for MIMD Multiprocessors. Technical Report PCS-TR94-226, Dept. of Computer Science, Dartmouth College, July 1994.

[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. In *AIAA 10th Computational Fluid Dynamics Conference*, June 1991.

[MLG92] T. Mowry, M. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. *Architectural Support for Programming Languages and Operating Systems*, pages 62–73, 1992.

[MS94] S. Moyer and V. Sunderam. PIOUS: A Scalable Parallel I/O System for Distributed Computing Environments. In *Proceedings of the Scalable High Performance Computing Conference*, May 1994.

[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] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *ACM SIGMOD Conference*, pages 109–116, June 1988.

[PGS93] D. Patterson, G. Gibson, and M. Satyanarayanan. A Status Report on Research in Transparent Informed Prefetching. Technical Report CMU-CS-93-113, Carnegie Mellon University, February 1993.

[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] R. 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. Available as NPAC Technical Report SCCS–633.

[SBW91] J. Saltz, H. Berryman, and J. Wu. Multiprocessors and Runtime Compilation. *Concurrency: Practice and Experience*, pages 573–592, December 1991.

[SD90] R. Schriber and J. Dongarra. Automatic Blocking of Nested Loops. Technical report, Research Institute for Advanced Computer Science, May 1990.

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

[TBC94a] 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*, pages 382–391, July 1994.

[TBC$^+$94b] R. Thakur, R. Bordawekar, A. Choudhary, R. Ponnusamy, and T. Singh. PASSION Runtime Library for Parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, October 1994.

[vKK$^+$92] R. von Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler Analysis for Irregular Problems in Fortran D. In *Proceedings of the* 5^{th} *Workshop on Languages and Compilers for Parallel Computing*, August 1992.

[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.

[WB87] M. Wolfe and U. Banerjee. Data Dependence and its Application to Parallel Processing. *International Journal of Parallel Programming*, 16(2):137–178, April 1987.

[WL91] M. Wolf and M. Lam. A Loop Transformation Theory and An Algorithm to Maximize Parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

[Wol87] M. Wolfe. Iteration space tiling for memory hierarchies. Extended version appeared in the Proceedings of the Third SIAM Conference on Parallel Processing, December 1987.

[Wol89a] M. Wolfe. More iteration space tiling. *Proceedings of Supercomputing'89*, November 1989.

[Wol89b] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambridge, MA, 1989.

[Wol92] M. Wolf. *Improving Locality and parallelism in Nested Loops.* PhD thesis, Stanford University, 1992. CSL-TR-92-538.

[ZBC$^+$92] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran - a Language Specification. Technical Report ICASE Interim Report 21, MS 132c, ICASE, NASA, Hampton VA 23681, 1992.