Dictionary on Parallel Input/Output

Master's Thesis

Heinz Stockinger

Advisor: ao.Univ.-Prof. Dr. Erich Schikuta

Institute for Applied Computer Science and Information Systems
Department of Data Engineering, University of Vienna
Rathausstr. 19/4, A-1010 Vienna, Austria
`heinz@vipios.pri.univie.ac.at`

February 9, 1998

To Christina

Acknowledgments

This is the place to say thank you to my advisor Erich Schikuta who has always been a very motivating and encouraging mentor. I also want to thank Thoms Mück who introduced me to the ViPIOS team after I had asked him the following simple question in one of his lectures "Is there a possibility of using parallel programming methods on a PC?". What is more, I have gained much experience concerning operating systems, and parallel and concurrent programming during a year of study in London (UK). Thank you to Gerald Quirchmayr, my ERASMUS coordinator who made possible my stay in England. The work in the research project was very incouraging because of an excellent team spirit: thanks to Thomas Fürle, Helmut Wanek and to my twin brother Kurt who have answered many of my questions patiently. Last but not least thank you very much to Horst Ebel who was a great help with the layout and the design of this dictionary. Some of the details in the dictionary have been discussed with the research teams themselves via questionaires. Thank you to all who have answered them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the last decades VLSI technology has improved and, hence, the power of processors. What is more, also storage technology has produced better main memories with faster access times and bigger storage spaces. However, this is only true for main memory, but not for secondary or tertiary storage devices. As a matter of fact, the secondary storage devices are becoming slower in comparison with the other parts of a computer.

Another driving forces for the development of big main memories are scientific applications, especially Grand Challenge Applications. Here a big amount of data is required for computation, but it does not fit entirely into main memory. Thus, parts of the information have to be stored on disk and transferred to the main memory when needed. Since this transfer - reading from and writing to disk, also known as input/output (I/O) operations - is very time consuming, I/O devices are the bottleneck for compute immense scientific applications. Recently much effort was devoted to I/O, in particular universities and research laboratories in the USA, Austria, Australia, Canada and Spain. There are many different approaches to remedy the I/O problem. Some research groups have established I/O libraries or file systems, others deal with new data layout, new storage techniques and tools for performance checking.

This dictionary gives an overview of all the available research groups in David Kotz's homepage (see below) and depicts the special features and characteristics. The *abstracts* give short overviews, *aims* are stated as well as *related work* and *key words.* The *implementa-*

tion platform shall state which platforms the systems have been tested on or what they are devoted to. *Data access strategies* illustrate special features of the treatment of data (e.g.: synchronous, locking, ...). *Portability* includes whether a system is established on top of another one (e.g.: PIOUS is designed for PVM) or if it supports interfaces to other systems. Moreover, many systems are applied to real world applications (e.g.: fluid dynamics, aeronautics, ...) or software has been developed (e.g.: matrix multiplication). All this is stated at *application*. Small examples or code fragments can illustrate the interface of special systems and are inserted at *example*.

What is more, some special features are depicted and discussed explicitly. For instance, a Two-Phase Method from PASSION is explained in detail under a specific key item. Furthermore, also platforms like the Intel Paragon are discussed. To sum up, the dictionary shall provide information concerning research groups, implementations, applications, aims, functionalities, platforms and background information. The appendix gives a detailed survey of all the groups and systems mentioned in the Dictionary part and compares the features.

This dictionary is intended for people having background information regarding operating systems in general, file systems, parallel programming and supercomputers. Moreover, much work is based on C, C++ or Fortran, and many examples are written in these programming languages. Hence, a basic knowledge of these languages and the general programming paradigms is assumed.

Most of the facts presented in this dictionary are based on a web page of David Kotz, Dartmouth College, USA, on parallel I/O. The internet made it possible to obtain all necessary information through ftp-downloading instead of using books or physical libraries (the web could be referred to as a logical library). This is also the place to thank David Kotz for his work, because without it, this dictionary would never have been written. The URL for the homepage is:

`http://www.cs.darmouth.edu/pario`

Note that most of the names listed at *people* are only from people participating in writing the papers, i.e. normally the names listed in the papers of the bibliography are listed in this work.

Layout and Text Style

Text in Helvitica 12 defines keywords that also contain a body of explanation. Text written in ***bold italics*** refers to another keyword (similar to a link in a markup language). **Simple bold** texts are important items or words which can be looked up in the dictionary, but which have no explanation body devoted to. Headlines within the bodies are always indicated by numbers followed by text in ***bold italics*** (e.g. ***2.1 Architectural Design***). There is no difference in the font size between headlines and sub headlines (e.g. ***1 Design***, ***1.1 Features***). Text in rectangular brackets refers to the text sources, e.g. [95].

Chapter 2

Dictionary

A

abort see ***concurrency algorithms, Strict Two-Phase Locking***

ACU (Array Control Unit) see ***CVL***

ADIO (Abstract-Device Interface for Portable Parallel-I/O)

abstract:

Since there is no standard ***API*** for parallel I/O, ADIO is supposed to provide a strategy for implementing ***API***s (not a standard) in a simple, portable and efficient way [144] in order to take the burden of the programmer of choosing from several different ***API***s. Furthermore, it makes existing applications portable across a wide range of different platforms. An ***API*** can be implemented in a portable fashion on top of ADIO, and becomes available on all file systems on which ADIO has been implemented (see Figure 2.1).

aims:

The main goal is to facilitate a high-performance implementation of any existing or new ***API*** on any existing or new file-system interface. The main objectives are:

- facilitate efficient and portable implementations of ***API***s

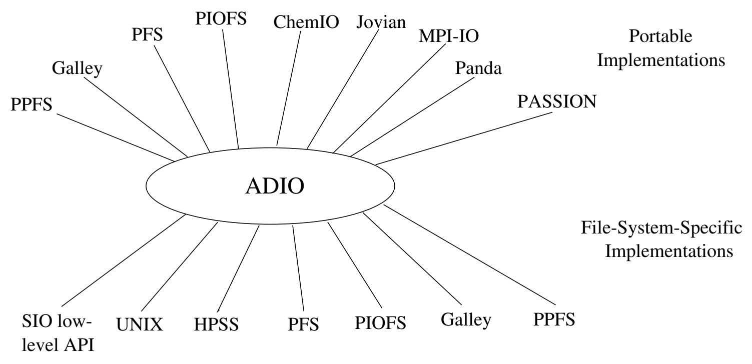


Figure 2.1: The ADIO Concept

- enable users to experiment with existing and new ***API***s
- make applications portable across a wide range of platforms

implementation platform:

The performance of ADIO was tested with two test programs and one real production parallel application on the **IBM SP** at ***Argonne National Laboratory*** and on the ***Intel Paragon*** at Caltech (both USA).

data access strategies:

File open and close are implemented as collective operations. Moreover, it provides routines for contiguous and **strided access** (see ***MPI-IO***), and makes use of ***MPI***'s **derived datatype**s. Read/write calls can be blocking, nonblocking or collective.

portability:

- ADIO was implemented on top of ***PFS*** and ***PIOFS***
- subsets of ***PFS***, ***PIOFS*** and ***MPI-IO*** were implemented on top of ADIO (logical views of ***PIOFS*** can be mapped to **derived datatype**s)
- ***Panda*** and ***PASSION*** can be implemented by using ADIO's routines for collective and **strided access**
- ADIO uses ***MPI*** wherever possible (e.g. **derived datatype**s)
- parameters like `disp, etype` and `filetype` from ***MPI-IO*** are employed

related work:

MPI-IO, MPI

ADIO is a key component of ***ROMIO*** (a portable implementation of ***MPI-IO***)

application:

Studying the nonlinear evolution of Jeans instability in self-gravitating gaseous clouds, which is a basic mechanism for the formation of stars and galaxies.

people:

Rajeev Thakur, William Gropp, Ewing Lusk

{thakur, gropp, lusk}@mcs.anl.gov

institution:

Mathematics and Computer Science Divison, ***Argonne National Laboratories***, S. Cass Avenue, Argonne, IL 60439, USA

http://www.mcs.anl.gov/scalable/scalable.html

key words:

device interface, ***API***

example:

ADIO instructions are very similar to ***MPI-2*** instructions (e.g.: `ADIO_Open` is similar to `MPI_Open`)

```
ADIO_File ADIO_Open (MPI_Comm comm, char *filename,
  void *file_system, int access_mode, ADIO_Offset disp,
  MPI_Datatype etype, MPI_Datatype filetype, int iomode,
  ADIO_Hints *hints, int perm, int *error)
```

ADOPT (A Dynamic scheme for Optimal PrefeTching in parallel file systems)

abstract:

ADOPT is a dynamic ***prefetching*** scheme that is applicable to any distributed system, but major performance benefits are obtained in **distributed memory** I/O systems in a parallel processing environment [133]. Efficient accesses and ***prefetching*** are supposed to be obtained by exploiting access patterns specified and generated from users or compilers.

aims:

Development of disk access strategies in concurrent/***parallel file system***s when the workload is expected to be generated by one or more parallel programs on a parallel machine [133].

implementation platform:

Intel iPSC/860

related work:

disk-directed I/O

people:

Travinder Pal Singh, Alok Choudhary

{tpsingh, choudhar}@cat.syr.edu

institution:

Department of Elect. and Comp. Engineering, Syracuse University, Syracuse, NY 13244, USA

keywords:

prefetching, cache

details:

I/O nodes are assumed to maintain a portion of memory space for ***caching*** blocks. This memory space is partitioned into a current and a prefetch cache by ADOPT. In particular, the current cache serves as a buffer memory between **I/O node**s and the disk driver whereas a prefetch cache has to save prefetched blocks. Prefetch information is operated and managed by ADOPT at the **I/O node** level.

[133] distinguishes between three levels in which access information from processes can be extracted:

1. The user embeds prefetch calls while programming.
2. The compiler extracts static access information.
3. Access information in form of prefetch requests is passed on to ADOPT during the execution of the processes.

The two major roles of the I/O subsystem in ADOPT are receiving all prefetch and disk access requests and generating a schedule for disk I/O. Finally, ADOPT uses an Access Pattern Graph (APGraph) for information about future I/O requests.

agent see ***Agent Tcl, PPFS , SHORE, TIAS***

Agent Tcl

abstract:

A transportable **agent** is a named program that can migrate from machine to machine in a heterogeneous network [61]. It can suspend its execution at an arbitrary point, transport to another machine and resume execution on the new machine. What is more, such **agent**s are supposed to be an improvement of the conventional **client-server** model.

aims:

Agent Tcl defines the following goals [62]:

- Reduced migration to a single instruction like the Telescript go, and allow this instruction to occur at arbitrary points.
- Provide transparent communication among **agent**s.
- Support multiple languages and transport mechanisms, and allow the straightforward addition of a new language or transport mechanism.
- Run on general UNIX platforms, and port as easily as possible to non-UNIX platforms.
- Provide effective security and fault tolerance.
- Be available in the **public domain**.

implementation platform:

Agent Tcl will run on standard UNIX platforms and supports multiple languages (**Tcl**, Java and Scheme) and transport mechanisms, but it is far away form being complete [61].

data access strategies:

Four basic communication behaviors:

1. naive: Each call is a blind request for data (single request to the server).
2. remote procedure call: An **agent** invokes a remote service (sequence of requests and responds).

3. remote evaluation: The **agent** sends a child to a remote machine. The child **agent** executes and returns its results.
4. migration: The **agent** migrates from machine to machine.

portability:

UNIX platforms in general

related work:

- AI (Artificial Intelligence): In AI an **agent** is an entity that perceives its environment with sensors and acts on its environment with effectors.
- Personal assistants: The **agent** is a program that relieves the user of a routine task.
- Distributed information retrieval: An **agent** searches multiple information resources for the answer to a user query. Each **agent** is responsible for searching the resources at its site and for communicating partial search to the other transportable **agent**s.
- software interoperation (An agent is a program that communicates correctly in some universal communication language such as KQML.)

application:

Agent Tcl is used in a range of information-management applications. It is also used in serial workflow applications (e.g. an **agent** carries electronic forms from machine to machine). It is used for remote salespersons, too.

people:

Robert S. Gray, George Cybenko, David Kotz, Daniela Rus

`{rgray, gvc, dfk, rus}@cs.dartmouth.edu`

institution:

Department of Computer Science, Dartmouth College, Hanover, NH 03755, USA

`http://www.cs.dartmouth.edu/agent/lab/`

key words:

client-server, migration, **RPC**, **Tcl**, Java, Scheme, mobile-**agent**

example:

The following example from [61] shows an **agent** that submits a child that jumps from machine to machine and executes the UNIX `who` command on each machine.

```
  # procedure WHO is the child agent that does the jumping
proc who machines {
  global agent
  set list ""

  # jump from machine to machine and exectute the UNIX who command
    on each machine
  foreach m $machines {
    if {catch "agent_jump" $M"} {
      append list "$m:\n unable to JUMP to this machine"
    else {
      set users [exec who]
      append list "agent (local-server):\n$users\n\n"
    }
  }
  return  $list
}

set machines "bald cosmo lost-ark temple-doom moose muir tanaya tioga
    tuolomne"
  # get name from server
agent_begin
  # submit the child agent that jumps
agent_submit $agent(local-ip) -vars machines -procs who -script {who
  $machines}
  # wait for and output the list of users
agent_receive code string -blocking
puts $string
  #agent is done
agent_end
```

possible ouput:

```
bald.cs.dartmouth.edu:
rgray ttyp2 Sep  5 21:24 (:0.0)
rgray tty6 Sep  7 07:14

cosmo.dartmouth.edu:

gvc pts/0 Aug 23 10:11
```

details:

1 Background

Agents are an extension of the **client-server** paradigm which divides programs into fixed rolls. The client can only execute services that are supported by the server (The most common communication mechanism is **message passing**), and the programmer is required to handle low-level details. **Remote Procedure Call (RPC)** hides theses low-level details by allowing a client to invoke a server operation using the standard procedure call mechanism. Similar to **client-server**, the client is limited to services supported by a server, therefore, transmitting the intermediate data is a waste of time. Avoiding this, the clients send a subprogram to the server. The subprogram executes at the server and returns only the final result to the client, and all intermediate data transfer is eliminated.

Transportable **agent**s are autonomous programs that communicate and migrate at will, support the **peer-to-peer** model and are either clients or servers. Furthermore, they do not require permanent connection between machines and are more fault-tolerant.

2 Architecture

The architecture has four levels: An ***API*** for each transport mechanism, a server that accepts incoming **agent**s and provides **agent** communication (send messages; an **agent** can send itself or a child **agent** to a remote site), an interpreter for each supported language, and on top the **agent**s themselves (see Figure 2.2). The interpreter consists of four components: an interpreter, a security module that prevents an **agent** from taking malicious actions, a state module that captures and restores the internal state of an executing **agent** and an ***API*** that interacts with the server to handle migration, communication and ***checkpointing***.

The second level (server or engine) performs the following tasks [63]:

- Status: Keep track of **agent**s running on the machine and answer queries about their status.

- Migration. Accept incoming **agent**s and pass the authenticated **agent** to the applica-

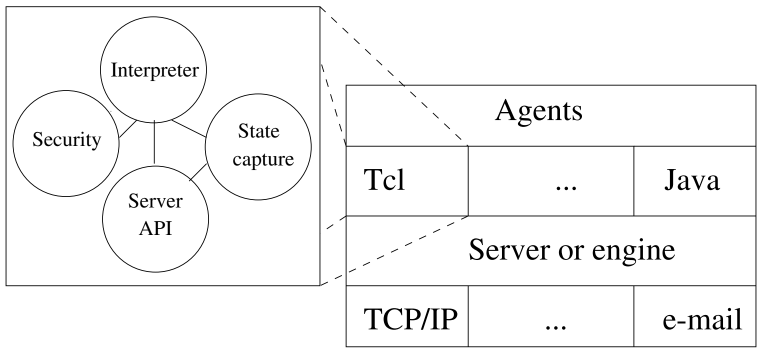


Figure 2.2: Architecture of Agent Tcl

tion interpreter.

- Communication: Allow **agent**s to send messages to each other within the namespace.
- Nonvolatile store

3 Agent Tcl Version 1.1

Tcl is a high-level **scripting language** (developed in 1987), is interpreted and so it is highly portable and easier to make sure. [60] states several disadvantages:

- inefficient compared to most other interpreted languages
- not object-oriented
- no code modularization aside from procedures
- difficult to write and debug large scripts

The architecture of Agent Tcl has two components: The server and a modified **Tcl** core with a **Tcl** extension. The server runs on each network site and is implemented as two components: the socket watcher for incoming **agent**s, messages and requests - and the **agent** table which keeps track of the **agent**s that are running on the machine and buffers incoming messages until the destination **agent** receives them.

all-to-all communication see ***Application Programming Interface***

Alloc Stream Facility (ASF)

Alloc Stream Facility (ASF) is an application-level I/O facility in the ***Hurricane File System*** (***HFS***). It can be used for all types of I/O, including disk files, terminals, pipes, networking interfaces and other low-level devices [96]. An outstanding difference to ***UNIX I/O*** is that the application is allowed direct access to the internal buffers of the I/O library instead of having the application to specify a buffer into or from which the I/O data is copied. The consequence is another reduction in copying.

ASC (Alloc Stream Interface) is used internally in ASF, but it is also made available to the application programmer. It is modeled in C. The most important operations in **ASI** are `salloc` and `sfree` which correspond to the operations `malloc` and `free` in C. Furthermore, **ASI** allows a high degree of concurrency when accessing a stream. The data is not copied from user buffer, and the stream only needs to be locked while the library's internal data structures are being modified. See also ***HFS***.

Alloc Stream Interface (ASI) see ***Alloc Stream Facility, HFS***

ANL (Argonne National Laboratory) - Parallel I/O Project

abstract:

ANL builds and develops a testbed and parallel I/O software and applications that will test and use this software. The I/O system applies two layers of high-performance networking. The primary layer is used for interconnection between **compute node**s and **I/O server**s whereas the second layer connects the **I/O server**s with ***RAID*** arrays.

aims:

- principal goal: create a testbed rich enough that a large number of experiments can be conducted without requiring new hardware to develop [68].
- Overall goal: develop the framework for creating balanced high-performance com-

puting systems in the future that are highly I/O capable [68]. A further aim is to provide the capability to test both parallel I/O systems and to validate those systems in the context of a robust mass storage environment [68].

implementation platform:

IBM SP1 (**compute nodes**), **IBM RS/6000** (**I/O servers**), **IBM 9570 RAID**

related work:

ANL collaborates with the National Storage Laboratory (NSL) and with **Scalable I/O Initiative**

people:

Mark Henderson, Bill Nickless, Rick Stevens

{henderson,nickless, stevens}@mcs.anl.gov

institution:

Mathematics and Computer Science Division, ANL, Argonne, IL 60439, USA

http://www.mcs.anl.gov/index.html

anonymous objects see ***SHORE***

AP1000 see ***Fujitsu AP1000***

API see ***Application Program Interface***

application process (A/P) see ***Jovian, ViPIOS***

Application Program Interface (API)

There are two complementary views for accessing an **OOC** data structure with a **global view** or a **distributed view** [9]. A **global view** indicates a copy of a globally specified subset of an **OOC** data structure distributed across disks. The library needs to know the ***distribution*** of the **OOC** and in-core data structures as well as a description of the requested data transfer. The library has access to the **OOC** and in-core data ***distribution***s. With a **distributed view**, each process effectively requests only the part of the data structure that it requires, and an exchange phase between the **coalescing processes** is needed (**all-to-all communication** phase). See also ***Jovian***.

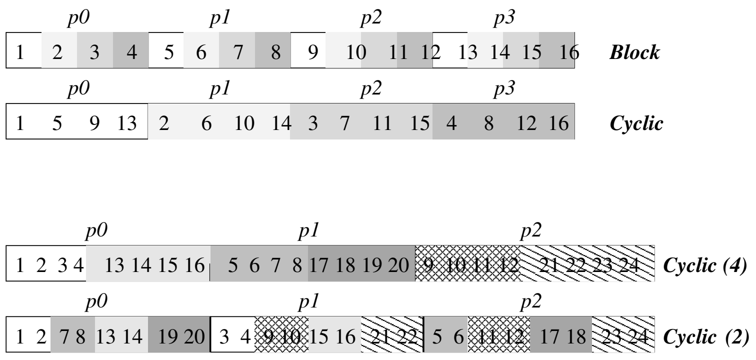


Figure 2.3: Redistribution Examples

array distribution

An array distribution can either be regular (block, cyclic or block-cyclic) or irregular (see ***irregular problems***) where no function specifying the mapping of arrays to processors can be applied [142]. Data that is stored on disk in an array is distributed in some fashion at compile time, but it does not need to remain fixed throughout the whole existence of the program. There are some reasons for redistributing data:

- Arrays and array sections can be passed as arguments to subroutines.
- If the ***distribution*** in the subroutine is different from that in the calling program, the array needs to be redistributed.

PASSION defines different forms of **redistribution** of data (see Figure 2.3): The **redistribution** of multidimensional arrays can be obtained in two forms:

- shape retaining: The shape of the local array remains unchanged.
- shape changing: The shape changes after **redistribution**.

When a main program calls a subroutine with a different ***distribution***, it is necessary to redistribute. After returning from the subroutine, it is necessary to redistribute (=**re-redistribution**) to the original format.

Array Control Unit (ACU) see ***CVL***

ASI (Alloc Stream Interface) see ***Alloc Stream Facility, HFS***

automatic data layout

A ***Fortran D*** program can be automatically transferred into an ***SPMD*** node program for a given **distributed memory** target machine by using data layout directives. A good data layout is responsible for high performance [91]. An algorithm partitions the program into code segments, called phases. Some case studies are presented in [80]. Remapping ideas between phases can be found in [92]. See also ***disk-directed I/O***.

B

BFS (Block File Server) see ***Hurricane File System (HFS)***

Bit-Interleaved Parity (RAID Level 3) see ***RAID***

Block-Interleaved Distributed-Parity (RAID Level 5) see ***RAID***

Block-Interleaved Parity (RAID Level 4) see ***RAID***

Block File Server (BFS) see ***Hurricane File System (HFS)***

block services see ***ParFiSys***

BNN Butterfly see ***Bridge parallel file system***

bottleneck see ***CHANNEL, file level parallelism, I/O problem, Pablo, parallel file system, RAID, Strict Two-Phase Locking***

Bridge parallel file system

This file system provides three interfaces from a high-level UNIX-like interface to a low-level interface that provides direct access to the individual disks [54]. A prototype was built on the **BNN Butterfly**. See also ***SPIFFI***.

Butterfly Plus see ***RAPID***

C

C*

C* supports ***data parallel*** programming where a sequential program operates on parallel arrays of data, with each virtual processor operating on one parallel data element [39]. A computer multiplexes physical processors among virtual processors to support the parallel model.

A variable in C* has a shape describing the rectangular structure and defining the logical configuration of parallel data in virtual processors. Since C* is restricted to in-core data, ***ViC**** is proposed to deal with **out-of-core** computation.

cache agent see ***Portable Parallel File System (PPFS)***

Cache Coherent File System (CCFS)

abstract:

The Cache Coherent File System (CCFS) is the successor of ***ParFiSys*** and consists of the following main components [24]:

- **Client File Server (CLFS)**: deals with user requests providing the file system functionality to the users, and interfacing with the CCFS components placed in the **I/O node**s; it is sited on the clients
- **Local File Server (LFS)**: interfaces with I/O devices to execute low level I/O and is sited on disk nodes
- **Concurrent Disk System (CDS)**: deals with **LFS** low level I/O requests and executes real I/O on the devices and is sited on the disk nodes

aims:

The objectives of the CCFS are: a cache coherent, scalable, concurrent file system providing a network of processors with a versatile access to file storage.

implementation platform:

GPMIMD

data access strategies:

see ***ParFiSys***

related work:

ParFiSys

people:

see ***ParFiSys***

institution:

see ***ParFiSys***

key words:

distributed memory, file system, **client-server**

details:

The **CLFS** is the first level within CCFS and its main function is to cache information. The models of a **LFS** as well as **CLFS** are related to a **client-server** model and include some main concepts: **light-weighted processes** and channels. **Light-weighted processes (LWP)** allow to treat several requests of the **CLFS** concurrently without blocking. Channels are used to transfer massive data in a **one-to-one communication**. The main loops look like that:

- wait for a message to arrive (communication is accomplished via **message passing**)
- extract the system call number and use it as an index into a function table
- call the appropriate procedure and wait for the answer (thread creation)
- reply a request

CDS provides parallel access to disk controllers and disk devices over a net of processors. The main concepts are high performance, **scalability**, ***distribution*** and concurrency. **CDS** will not directly be seen by the users, but a file system or a database will be between users and **CDS**.

1 Design Features of CDS

- Since each type of disk offers a different balance of fault-tolerance, performance and cost, **CDS** should satisfy a wide range of applications and interact with different disks.
- It should provide two different services (local and distributed), but with the same interface.
- Process communication is handled by a **POSIX** like **message passing** mechanism. Primitive send and receive commands are used, and reliable communication is assumed.
- Processes and threads will be used for process management. Each heavyweight process will have many threads executing concurrently.
- Consistency always has to be insured by locking of data blocks.
- **CDS** carries out the mapping from virtual to physical blocks. The virtual address is passed to a **CDS** routine which maps it to the physical address and carries out the corresponding I/O operation. **Scalability** is enhanced by adding disk nodes dynamically to the I/O system.

2 CDS Architecture

CDS is similar to a **client-server** model, but in a distributed fashion. Each process can be referred to as a client of a big unique ”virtual disk” server. Moreover, **CDS** heavily uses threads (in order to allow the server to treat several requests concurrently without blocking), ports (in order to receive requests from clients) and channels (transfer data in a **one-to-one communication**).

2.1 Server

CDS does not have just a unique, but one server per disk node. The main goal of the single servers is to exploit concurrency in the access to its node. Main features [125]:

- It is very simple. Only basic single node access of other **CDS** servers.
- Is does not worry about the existence of other **CDS** servers.

- Is does not worry about the birth and death of the clients.
- It is only devoted to serve requests with local blocks.
- It serves only self-contained requests.
- It is usually a very long life and robust process.

Each **CDS** server has an own block cache in order to be efficient enough. Moreover, **Last Recently Used** policy is used.

2.2 Client

Clients have to be able to distribute incoming requests between several servers. The main features are [125]:

- It is not a simple **RPC** interface, but it implements part of the whole **CDS** functionality.
- It has to keep information about how many **CDS** servers exist and where they are.
- Is does not worry about the birth and death of the clients.

cache manager see ***Galley***

caching

In order to avoid or reduce the latency of physical I/O operations, data can be cached for later use. Read operations are avoided by ***prefetching*** and write operations by postponing or avoiding writebacks. Additionally, smaller requests can be combined and large requests can be done instead. One important question is the location of the cache. ***PPFS*** employs three different levels: server cache (associated with each I/O sever), client cache (holds data accessed by user processes), and global cache (in order to enforce consistency) [73].

CAP Research Program

CAP is an integral part of parallel computing research at the Australian National University (ANU). A great deal of work is devoted to the ***Fujitsu AP1000*** since CAP is an agreement between ANU and the High Performance group of Fujitsu Ltd.

CC++ (Concurrent C++) see ***task-parallel program***

CCFS see ***Cache Coherent File System***

CDS (Concurrent Disk System) see ***Cache Coherent File System (CCFS)***

CDS (Client-distributed state) see ***concurrency algorithms***

cell see ***Vesta***

CFS see ***Concurrent File System***

CHANNEL

PASSION introduces CHANNEL as modes of communication and synchronization between ***data parallel*** tasks [4]. A CHANNEL provides a uniform one-directional communication mode between two ***data parallel*** tasks, and concurrent tasks are plugged together. This results in a **many-to-many communication** between processes of the communicating tasks. The general semantics of a CHANNEL between two tasks are as follows:

- Distribution Independence: If two tasks are connected via a CHANNEL, they need not have the same data ***distribution***, i.e. whereas task 1 employs a cyclic fashion, task 2 can use a block fashion and the communication can still be established. Hence, both data ***distribution***s are independent.
- Information Hiding: A task can request data from the CHANNEL in its own ***distribution*** format. This is also true if both tasks use different data ***distribution*** formats.
- Synchronization: The task wanting to receive data from the CHANNEL has to wait for the CHANNEL to get full before it can proceed.

PASSION offers two different approaches to implement a CHANNEL: **Shared File Model** and **Distributed File Model**.

1 Shared File Model (SFM)

Such a CHANNEL uses a shared file and is uni-directional. The shared file consists of many regions which may be contiguous or striped. It has the following characteristics:

- Multiple-Readers/Multiple-Writers are allowed, but the compiler has to generate the correct code to avoid overwriting of data. Furthermore, each reading process has its own file pointer which can read any part of the file.

- Synchronization is gained via a synchronization variable at the beginning of the file.

The advantage of **SFM** is that it allows the communication of a dissimilar set of processors and, hence, extends easily to a heterogeneous environment [4]. They can even have their own file systems as long as they are convertable to the other's machine. However, a single file can also be a **bottleneck** in the system, and performing read/write to synchronize may be time consuming.

2 Multiple File Model (MFM)

In contrast to **SFM**, **MFM** allows data to be communicated in a pipelined fashion by breaking the data structure into a number of smaller data sets. The intermediate storage facility consists of multiple files. Unfortunately, these multiple files introduce extra overheads for the implementation [4].

CHAOS

abstract:

CHAOS deals with efficiently coupling multiple ***data parallel*** programs at runtime. In detail, a mapping between data structures in different ***data parallel*** programs is established at runtime. Languages such as ***HPF***, C an **pC++** are supported [123]. The approach is supposed to be general enough for a variety of data structures.

aims:

coupling of multiple ***data parallel*** programs at runtime

The primary goals of the implementation were language independence, flexibility and efficiency.

implementation platform:

The implementation runs on a cluster of four-processor **SMP Digital Alpha** Server 4/2100 symmetric processors, and the nodes are connected by an FDDI network.

data access strategies:

Efficiency is obtained by buffering and asynchronous transfer of data as well as precomputation of optimized schedules. The programming model provides two primary operations: exporting individual arrays and establishing a mapping between a pair of exported arrays.

portability:

PVM is used for establishing the underlying messaging layer.

related work:

similar to the software bus approach used in Polith [123]

single address space operating system **Opal**

Linda could also be used to couple programs [123]

Meta-Chaos, ***PARTI***

peope:

Guy Edjlali, Alan Sussman, Joel Saltz, M. Ranganathan, A. Acharaya

{edjlali, als, saltz, ranga, acha}@cs.umd.edu

institution:

Department of Computer Science and UMICAS, University of Maryland, College Park, MD 20742, USA

http://www.cs.umd.edu/projects/hpsl.html

key words:

coupling, **distributed memory**

example:

An example of such a mapping can look like follows: A pair of simulations working on neighboring grids and periodically exchanging data at the boundary. Here the array sections in both programs which correspond to the common boundary would be mapped together [49].

program source

```
integer, dimension (200,199)::B
integer, dimension (2)::Rleft, Rright

Rleft (1) = 50    Rleft(2) = 50
Rright (1) = 100  Rright (2) = 100
```

```
regionId=CreateRegion_HPF (2, Rleft(1), Rright(1))
src_setOfRegionId = MC_NewSetOfRegion ()
MC_AddRegion2Set (RegionId, src_setOfRegionId)
schedule = MC_ComputeSched (HPF, B, src_setOfReginId)
call MC_DataMoveSend (schedId, B)
```

program destination

```
integer, dimension (50,60)::A
integer, dimension (2)::Rleft, Rright

Rleft (1) = 1    Rleft(2) = 10
Rright (1) = 50  Rright (2) = 50
regionId=CreateRegion_HPF (2, Rleft(1), Rright(1))
dest_setOfRegionId = MC_NewSetOfRegion ()
MC_AddRegion2Set (RegionId, dest_setOfRegionId)
schedule = MC_ComputeSched (HPF, B, dest_setOfReginId)
call MC_DataMoveSend (schedId, A)
```

Here two arrays are defined and the communication between them is established with the **Meta-Chaos** command `MC_DataMove`.

details:

Firstly, the implementation used asynchronous, one sided **message passing** for inter- application data transfer with the goal to overlap data transfer with computation [123]. Secondly, optimized messaging schedules were used. The number of messages transmitted has to be minimized. Finally, buffering was used to reduce the time spent waiting for data. The data transfer itself can be initiated by a consumer or a producer ***data parallel*** program. Furthermore, the inter-application data transfer is established via a library called **Meta-Chaos**. ***PVM*** is the underlying messaging layer, and each ***data parallel*** program is assigned to a distinct ***PVM*** group.

Meta-Chaos is established to provide the ability to use multiple specialized parallel libraries and/or languages within a single application [49], i.e. one can use different libraries in one program in order to run operations on distributed data structures in parallel.

Meta-Chaos introduces three techniques for allowing ***data parallel*** programs to interoperate:

1. Find out the unique features of both libraries and implement those in a single integrated runtime support library.
2. Use a costume interface between each pair of ***data parallel*** libraries.
3. Define a set of interface functions that every ***data parallel*** library must export, and build a so-called meta-library.

The CHAOS group is also involved in solving ***irregular problems***.

CHARISMA (CHARacterize I/O in Scientific Multiprocessor Applications)

abstract:

CHARISMA is a project to characterize I/O in scientific multiprocessor applications from a variety of production parallel computing platforms and sites [89]. It recorded individual read and write requests in live, multiprogramming workloads. Some results are presented in [89, 122]. It turned out that most files were accessed in complex, highly regular patterns.

aims:

measure real file system workloads on various production parallel machines

implementation platform:

CHARISMA characterized the file system activities of an ***Intel iPSC/860*** and a TMC **CM-5** [85].

people:

David Kotz, Nils Nieuwejar, A. Purakayastha, Carla Ellis, Michael L. Best

`{dfk, nils}cs.dartmouth.edu`

`apu@watson.ibm.com, carla@cs.duke.edu, mike@media.mit.edu`

institution:

- Department of Computer Science, Dartmouth College, Hanover, NH 03755-3510, USA
- Media Lab, MIT, Cambridge, MA 02139, USA
- Department of Computer Science, Duke University, Durham, NC 22708, USA

`http://www.cs.dartmouth.edu/research/charisma.html`

key words:

distributed memory, I/O characterization

check disk see ***coding techniques***

checkpointing

Checkpointing allows processes to save their state from time to time so that they can be restarted in case of failures, or in case of swapping due to resource allocation [10]. What is more, a checkpointing mechanism must be both space and time efficient. Existing checkpointing systems for **MPP**s checkpoint the entire memory state of a program. Similarly, existing checkpointing systems work by halting the entire application during the construction of the checkpoint. Checkpoints have low latency because they are generated concurrently during the program's execution. See also ***ChemIO***.

ChemIO (Scalable I/O Initiative)

abstract:

ChemIO is an abbreviation for High-Performance I/O for Computational Chemistry Applications. The **Scalable I/O Initiative** will determine application program requirements and will use them to guide the development of new programming language features, compiler techniques, system support services, file storage facilities, and high performance networking software [6] [34].

Key results are:

- implementation of scalable I/O algorithms in production software for computational chemistry applications
- dissemination of an improved understanding of scalable parallel I/O systems and algorithms to the computational chemistry community

aims:

The objectives of the Application Working Group of a the **Scalable I/O Initiative** include [121]:

- collecting program suites that exhibit typical I/O requirements for ***Grand Challenge Applications*** on massively parallel processors
- monitoring and analyzing these applications to characterize parallel I/O requirements for large-scale applications and establish a baseline for evaluating the system software and tools developed during the **Scalable I/O Initiative**
- modifying, where initiated by the analysis, the I/O structure of the application programs to improve performance
- using the system software and tools from other working groups and representing the measurement and analysis of the applications to evaluate the proposed file system, network and system support software, and language features
- developing instrumented parallel I/O benchmarks

application:

The Application Working Group has identified 18 major application program suites for investigating during the **Scalable I/O Initiative**.

- Biology
  - General Neuronal Simulation Systems (GENESIS)
  - 3-D Atomic Structure of Viruses
- Chemistry
  - Scalable I/O for Hartree-Fock Methods
  - Quantum Chemical Reaction Dynamics
  - Cross-Sections for Electron-Molecule Collisions
  - Electronic Structures for Superconductors
- Earth Sciences
  - Parallel NCAR Community Climate Model
  - Four Dimensional Data Assimilation
  - Land Cover Dynamics
  - Data Analysis and Knowledge Discovery

  - Scalable I/O for Scientific Data Processing
  - Massive SAR Processing of Large Radar Data Sets
- Engineering
  - 3-D Navier-Stokes
  - Exflow: a Compressible Navier-Strokes Solver
  - Scattering and Radiation from large Structures
- Graphics
  - I/O Support for Parallel Rendering Systems
- Physics
  - Vortex Models for High Tc Superconductors
  - Very Large FFT's - High Speed Data Acquisition

implementation platform:

Sun **SPARC**, DEC, HP 700 series, Intel Gamma, ***Intel Touchstone Delta***, **SGI**, ***Cray C-90***, Cray T-3D, **Cray Y-MP**, ***Intel Paragon***, **IBM SP1**, ***Intel iPSC/860***, ***nCUBE***-2, **CM-5**, **IBM RS6000**

people:

James C.T. Pool (Chair), Leon Alkalaj, Upinder S. Bhalla, David Bilitch, James Bower, C. Cohn, David W. Curkendall, Larry Davis, Erik De Schutter, Robert D. Ferrora, Ian Foster, G. Fox, Nigel Goddard, Par Hanarahn, Rick A. Kendall, Aron Kuppermann, Gary K. Leaf, Anthony Leonard, David Levine, Peter M. Lyster, Dan C. Marinescu, Vincent McKoy, Daniel I. Meiron, Edmond Mesrobian, John Michalakes, K. Mills, M. Minkoff, Caharles C. Mosher, Richard Muntz, Jean E. Patterson, Thomas A. Prince, Joel H. Saltz, Herbert L. Siegel, Paul E. Stolorz, Roy D. Williams, Carl L. Winstead, Mark Wu

institution:

ARCO Exploration and Production Technology, TX 75075

Argonne National Laboratory, IL 60439

California Institut of Technology, Pasadena, CA 91125

Gordon Space Flight Center

Jet Propulsion Laboratory, MS 198-219

Mt. Sinai School of Medicine, NY 10029

Parcific Northwest Laboratory, WA 99352-0999

Pittsburgh Supercomputing Center, PA 15213

Princton University, NJ 08544

Purde University, IN 47907

Syracuse University, NY 13244

University of California, CA 90024-1596

University of Maryland, MD 20742

`http://www.mcs.anl.gov/chemio/`

circular wait see ***Strict Two-Phase Locking***

Client-distributed state (CDS) see ***concurrency algorithms***

Client File Server (CLFS) see ***Cache Coherent File System (CCFS)***

client-server see ***Agent Tcl, Cache Coherent File System, EXODUS, Galley, object-oriented database, PIOUS, Portable Parallel File System (PPFS), SPFS, TOPs, Vesta, ViPIOS***

clustering

A file is divided into segments which reside on a particular server. This can be regarded as a collection of records. What is more, each file must have at least one segment [73]. See also ***PPFS***.

CM-2 (Connection Machine)

CM-2 is a ***SIMD*** machine where messages between processors require only a single cycle [97]. See also Appendix: Parallel Computer Architecture.

CM-5 see ***CHARISMA, ChemIO, PASSION, Multipol, CMMD I/O System, Appendix: Parallel Computer Architecture***

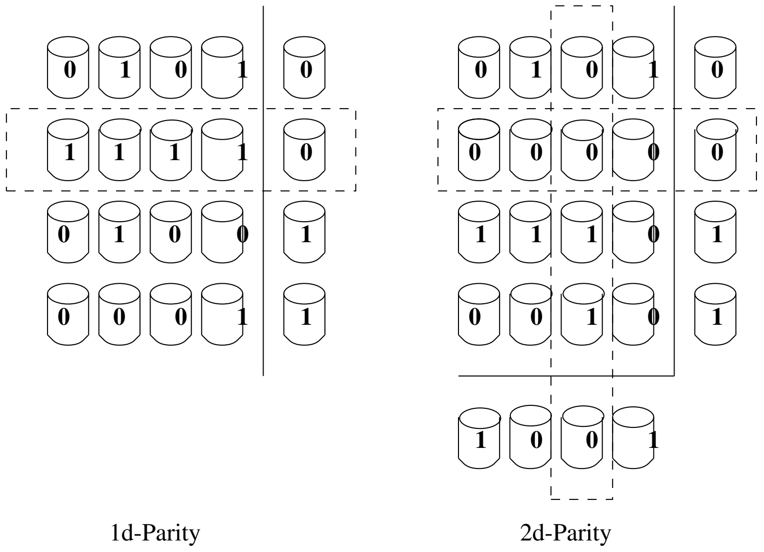


Figure 2.4: Parity examples

CMMD I/O System

This system provides a **parallel I/O interface** to parallel applications on the Thinking Machines **CM-5**, but the **CM-5** does not support a ***parallel file system***. Hence, data is stored on large high-performance ***RAID*** systems [54].

coalescing processes (C/P) see ***Jovian, Application Program Interface***

coding techniques

Magnetic disk drives suffer from three primary types of failures: transient or noise-related failures (corrected by repeating the offending operation or by applying per sector error-correction facilities), media defect (usually detected and corrected in factories) and catastrophic failures like head crashes. Redundant arrays can be used to add more security to a system. The scheme is restricted to leave the original data unmodified on some disks (**information disk**s) and define redundant encoding for that data on other disks (**check disk**s).

In the first example in Figure 2.4, the parity is always in the disk on the right side. The overhead is 1/G (where G is the number of disks) and the update penalty is 1 (one **check disk** update is required for every **information disk** update). The second example is an extension of the first one and uses a 2d-parity. On the end of each row and each column a **check disk** stores the parity. The overhead is $2G/G^2 = 2/G$ and the update penalty is 2 [67]. See also ***RAID***.

collective-I/O see ***disk-directed I/O, Jovian, Panda, PASSION, Vesta***
collective-I/O interface see ***disk-directed I/O, Panda***
collective communication see ***Jovian, loosely synchronous, MPI, PASSION***
commit see ***concurreny algorithms, PIOUS, Strict Two-Phase Locking***
communication coalescing see ***PARTI***
communicator see ***MPI, MPI-IO, PVM***
compute node see ***ANL, CFS, disk-directed I/O, Intel iPSC/860 hypercube, nCUBE, Panda, PIOFS, PIOUS, PASSION, Vesta***
compute processor see ***Galley***

concurrency algorithms

concurrency algorithms can be divided into two classes [108]:

- **Client-distributed state (CDS)** algorithms are optimistic and allow the **I/O daemon**s to schedule in parallel all data accesses which are generated by a given file pointer. This method can lead to an invalid state that forces rollback: a file operation may have to be abandoned and re-tried. **CDS** algorithms distribute global state information in the form of an operation "**commit**" or "**abort**" message, sent to the relevant **I/O daemon**s by the client. In ***PIOUS*** this model is realized with a transaction called volatile. **CDS** algorithms provide the opportunity to efficiently multicast global state information.

- **Server-distributed state (SDS)** algorithms are conservative, allowing an **I/O daemon** to schedule data access only when it is known to be consistently ordered with other data accesses. **SDS** never leads to an invalid state, because global state information is distributed in the form of a token that is circulated among all **I/O daemon**s servicing a file operation.

concurrency control

Sequential consistency (**serializability**) dictates that the results of all read and write operations generated by a group of processes accessing storage must be the same as if the operations had occurred within the context of a single process [108]. It should gain the effect of executing all data accesses from one file operation before executing any data accesses from the other one . This requires global information: each **I/O daemon** executing on each **I/O node** must know that it is scheduling data access in a manner that is consistent with all other **I/O daemon**s. Concurrency control algorithms can be divided into two classes: client-distributed and server-distributed. See also ***concurrency algorithms***.

Concurrent Disk System (CDS) see ***Cache Coherent File System (CCFS)***

Concurrent File System (CFS)

CFS is the file system of the ***Intel Touchstone Delta*** and provides a UNIX view of a file to the application program [15]. Four I/O modes are supported:

- Mode 0: Here each node process has its own file pointer. It is useful for large files to be shared among the nodes.
- Mode 1: The **compute node**s share a common file pointer, and I/O requests are serviced on a first-come-first-serve basis.
- Mode 2: Reads and writes are treated as global operations and a global synchronization is performed.
- Mode 3: A synchronous ordered mode is provided, but all write operations have to be of the same size.

conflict see ***Strict Two-Phase Locking*** control thread see ***SPIFFI***

CP (compute processor) see ***Galley***

CP Thread see ***Galley***

Cray C90

The Cray C90 is a **shared memory** platform [97].

Cray Y-MP see ***ChemIO, supercomputing applications***

CVL (C Vector Library)

abstract:

CVL (also referred to as **DartCVL**) is an interface to a group of simple functions for mapping and operating on vectors [38]. The target machine is a ***SIMD*** computer. In other words, CVL is a library of low-level vector routines callable from C [38].

aims:

The aim of CVL is to maximize the advantages of hierarchical virtualization.

implementation platform:

DEC 12000/Sx 2000 (equivalent to **MasPar MP-2** massively parallel computer)

related work:

UnCVL at University of North Carolina; there are two major differences between **DartCVL** and **UnCVL**:

1. **DartCVL** uses hierarchincal virtualization; **UnCVL** cut-and-stack
2. **DartCVL** runs as much serial code as possible; **UnCVL** runs all serial code on the **Array Control Unit (ACU)**

people:

Thomas Cormen, Nils Nieuwejar, Sumit Chawla, Preston Crow, Mlissa Hrischl, Roberto Hoyle, Keith D. Kotay, Rolf H. Nelson, Scott M. Silver, Michael B. Taylor, Rajiv Wickremesinghe

`{thc, nils}@cs.dartmouth.edu`

institution:

Department of Computer Science, Dartmouth College, Hanover, NH 03755, USA

`http://www.cs.dartmouth.edu/research/pario.html`

key words:

I/O interface, vector operations

D

DAC (Directed Acyclic Graphs) see ***SPFS***

Data Arrays are In-Core and Indirection Arrays of Out-of-core (DAI/IAO) see ***irregular problems***

DartCVL see ***CVL***

data declustering see ***Galley, parallel file system, PIOUS, RAID, ViPIOS***

data parallel

A data parallel program applies a sequence of similar operations to all or most elements of a large data structure [4]. ***HPF*** is such a language. A program written in a data parallel style allows advanced compilation systems to generate efficient code for most **distributed memory machines** [91]. Additionally, [91] provides some guidelines to write programs in a data parallel programming style:

- do not write machine-dependent code
- name data objets explicitly
- avoid indirect addressing where possible

data prefetching

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 request for the next **slab** (see ***PASSION***) immediately after the current **slab** has been read. As for prefetching, data is prefetched from a file, and on performing the computation on this data the results are written back to disk [33]. This is repeated again afterwards.

[82] studies prefetching and ***caching*** strategies for multiple disks in the presence of application-provided knowledge of future accesses. Furthermore, the tradeoffs between aggressive prefetching and optimal cache replacement are discussed. The results of a trace-driven simulation study of integrated prefetching and ***caching*** algorithms are presented. Following algorithms are discussed: aggressive, fixed horizon, reverse aggressive and forestall. Prefetching can preload the cache to reduce the cache miss ratio, or reduce the cost of a cache miss by starting the I/O early [116].

data reuse

The data already fetched into main memory is reused instead of read again from disk. As a result, the amount of I/O is reduced [33]. See also ***PARTI***.

data sieving

Normally data is distributed in a **slab** (see ***PASSION***) and not concentrated on a special address. Direct reading of data requires a lot of I/O requests and high costs. Therefore, a whole **slab** is read into a temporary buffer and the required data is extracted from this buffer and placed in the **ICLA** (see ***PASSION***).

All routines support the reading/writing of regular sections of arrays which are defined as any portion of an array that can be specified in terms of its lower bound, upper bound and stride in each dimension. For reading a strided section, instead of reading only the requested elements, large contiguous chunks of data are read at a time into a temporary buffer in main memory [33]. This includes unwanted data. The useful part of it is extracted from the buffer and passed to the calling program. A disadvantage is the high memory requirement for the buffer.

data striping see ***RAID***

DDLY (Data Distribution Layer)

abstract:

DDLY is a run-time library providing a fast high-level interface for writing parallel programs. DDLY is not yet ready, but is supposed to support automatic and efficient data partitioning and ***distribution*** for ***irregular problems*** on **message passing** environments. Addidionally, parallel I/O for both regular and ***irregular problems*** should be provided.

aims:

provide a smart interface for managing matrices as abstract objects

support I/O, partitioning and ***distribution***

data access strategies:

I/O routines provide independence between the formats of data stored in files and in memory. Formats such as Harwell-Boeing, typical raw and ASCII proprietary formats are supported. Furthermore, several representations of matrices in memeory can be managed (e.g. dense matrix, compressed sparse matrix).

portability:

HPF, ***Vienna Fortran***

key words:

I/O library, ***irregular problems***

people:

G. P. Trabada, E. L. Zapata

institution:

University of Malaga, Spain

example:

In the following program fragment from [147] (designed in a ***SIMD*** style) data is read from secondary storage and is distributed in an optimal way afterwards. Finally, each node operates on a local problem with local data.

```
#include <ddly.h>
void main
```

```
{
  ddly_m_descriptor *md;
  ddly_t_descriptor *td;
  int   n_dim, dim_size [MAX_DIM],
    Columns[],
        Rows[];
  float Values;

  // describe the current topology
  td = ddly_new_technology (n_dim, dim_size);

  //open matrix descriptor associated to file
  md = ddly_open_matrix (filename, DDLY_MF_HARWELLB);

  // read matrix data from file
  ddly_read_whole_matrix (md);

  // compute partition, \wrf{distribution} for matrix and distribute
  // data to all nodes
  ddly_mrd (md, td);

  // each node has now only local data
  Values = ddly_values (md);
  Columns = ddly_sp_columns (md);
  Rows = ddly_sp_rowpointers (md);

  // local computaion

  // close matrix descriptor
  ddly_close_matrix (md);
}
```

details:

The first part that has been developed for the mapping process is the Data Distribution Layer (DDLY) which hides the communiction required to distribute data on **distributed memory machine**s and the low level I/O operations. [147] distingushes between **virtual data structure (VDS)** and **internal data structure (IDS)**. These can be used for mapping the original code into the underlying data structure provided by run-time support. The I/O interface is similar to the ***UNIX I/O***, e.g. data structures can be managed in a similar way to UNIX file descriptors. In particular, a data ***distribution*** routine takes two arguments: a matrix descriptor and a topology descriptor.

deadlock see ***Hurricane operating system, Strict Two-Phase Locking***

DEC 12000/Sx 2000 see ***CVL***

DEC 3000/500 see ***SPFS***

DEC-5000 see ***disk-directed I/O, STARFISH***

DEC Alpha see ***TIAS, TPIE***

DEC MIPS see ***TIAS***

declustering see ***design of parallel I/O software, RAID***

delayed write see ***supercomputing applications***

demand-driven see ***in-core communication***

derived datatype see ***MPI, MPI-IO***

design of parallel I/O software

[40] uses theoretical and practical results to give some hints or guidelines for the design of parallel I/O software. Since the reason for the use of parallel machines is speed, the design must be performance oriented and scalable. Scalable means that the disk usage is asymptotically optimal as the problem and machine size increase. What is more, [40] describes two different uses of parallel I/O: A traditional file access paradigm where a program explicitly reads input files and writes output files, and the **out-of-core** one which is also known as **extended memory**, **virtual memory** or **external computing**. Here data does not fit entirely into main memory and is transferred from disk to memory and vice versa.

1 Necessary capabilities

[40] states that specific capabilities have to be fulfilled in order to support the most efficient parallel I/O algorithms. To start off, since a block contains the smallest amount of data that can be transferred in a single disk access [40], I/O operations always handle data which occurs in multiples of the block size. The requirements presented below apply to both ***SIMD*** (the controller organizes the disk access on behalf of the processors) and ***MIMD*** (the processors organize their own disk accesses) systems:

- control over **declustering**: **Declustering** means distributing data in each file across multiple disks [40]. Such a **declustering** is defined by two components, a **striping unit** and the **distribution pattern**. A **striping unit** refers to logically and physically contiguous data. A common **distribution pattern** is **round-robin**. These components should be changeable by the programmers individually.

- querying about the configuration: Get information about the number of disks, block size, number of processors, amount of available physical memory, etc.

- independent I/O: In contrast to fully striped access, where all blocks are accessed at the same location on each disk, blocks are accessed at different locations in an independent I/O model.

- bypassing parity

- turning off file ***caching*** and ***prefetching***

2 Interface Proposal

[83] proposes some features for an interface for parallel programming:

- Multiopen: A file must be opened for all parallel processes rather than open the file independently for each process. In other words, whenever a process opens a file, it is opened for the entire parallel application. Consequently, each file is only opened once, and each process has its own file descriptor. Additionally, a multiopen can even create a file if it does not exist.

- directory structure: A single file-naming directory should be used for the entire file system.

- file pointer: A file pointer can either be global or local when using a **multifile**.

- mapped file pointers

See also ***disk-directed I/O***.

DFP (distribution file pointer thread) see ***SPIFFI***

DH (Dirty Harry) see ***Hurricane File System (HFS)***

Direct Access see ***Portable Parallel File System (PPFS)***

Directed Acyclic Graphs (DAC) see ***SPFS***

Direct File Access see ***PASSION***

Dirty Harry (DH) see ***Hurricane File System (HFS)***

disk-directed I/O

abstract:

Disk-directed I/O can dramatically improve the performance of reading and writing large, regular data structures between **distributed memory** and distributed files [86], and is primarily intended for the use in multiprocessors [87].

implementation platform:

Examples are running on ***STARFISH*** (on top of Proteus parallel architecture simulator) on **DEC-5000** workstations

data access strategies:

Double-buffering and special remote memory messages are used to pipeline the data transfer. The key idea is that the data is moved between **compute nodes** and **I/O nodes**, and between the **I/O nodes** and the disk, in a schedule that is most optimal for the disks.

related work:

TPIE; Sandia National Laboratory produces a library of C-callable, low-level, synchronous I/O functions to run on the ***Intel Paragon*** under the SUNMOS operating system on **compute node**s and ***OSF/1*** on **I/O node**s; ***Panda***, ***nCUBE***, ***Vesta***

application:

Disk-directed I/O was effectively tested for the use of an **out-of-core** LU-decomposition computation [85].

people:

David Kotz

`{dfk}@cs.dartmouth.edu`

institution:

Department of Computer Science, Dartmouth College, Hanover, NH 03755-3510

`http://www.cs.darmouth.edu/research/pario.html`

key words:

distributed memory, ***prefetching***, **collective-I/O**, **strided access**, **nested patterns**

details:

In a traditional UNIX-like interface, individual processors make requests to the file system, even if the required amount of data is small. In contrast, a **collective-I/O interface** supports single joint requests from many processes. Disk-directed I/O can be applied for such an environment. In brief, a collective request is passed to the **I/O processor**s for examining the request, making a list of disk blocks to be transferred and sorting the list. Finally, they use double-buffering and special remote memory messages to pipeline the data transfer. This strategy is supposed to optimize disk access, use less memory and has less CPU and **message passing** overhead [88]. The potential for disk-directed I/O is explored in three situations in [86]: data-dependent ***distribution***, data-dependent filtering and irregular subsets (see ***irregular problems***).

[113] distinguishes between a sequential request to a file, which is at a higher offset than the previous one, and a consecutive request, which is a sequential request that begins where the previous one ended. Furthermore, termini like **strided access** and **nested patterns** are introduced. In a **simple-strided access** a series of requests to a node-file is done where each request has the same size and the file pointer is incremented by the same amount between each request. Indeed, this would correspond to reading a column of data from a matrix stored in row-major order [113]. A group of requests that is part of this **simple-strided** pattern is defined as a **strided segment**. **Nested patterns** are similar to **simple strided access**, but it is composed of **strided segment**s separated by regular strides in the file [113].

disk level parallelism see ***file level parallelism, HiDIOS***

disk manager see ***Galley***

Disk Resistent Arrays (DRA)

abstract:

DRA extend the programming model of ***Global Arrays*** to disk. The library contains details concerning data layout, addressing and I/O transfer in disk array objects. The main difference between DRA and ***Global Arrays*** is that DRA reside on disk rather than in main memory.

implementation platform:

Intel Paragon (PFS), ***IBM SP2*** (***PIOFS*** and local disks), Cray T3D, ***KSR-2***, SGI Power Challenge and networks of workstations

institution:

joint project between:

Pacific Northwest National Laboratory

Argonne National Laboratory

ChemIO

application:

DRA can be used to implement user-controlled virtual memory or checkpointing of programs that use distributed arrays.

distributed array see ***irregular problems, PARTI***

Distributed File Model see ***CHANNEL***

distributed memory see ***ADOPT, automatic data layout, CCFS, CHAOS, CHARISMA, data parallel, DDLY, disk-directed I/O, Global Arrays, HiDIOS, Intel iPSC/860 hypercube, Intel Paragon, irregular problems, Jovian, MIMD, MPI, MPI-2, Multipol, Pablo, Panda, PARTI, PRE, PASSION, PIOUS, PPFS, SPIFFI, Vesta, Vulcan multicomputer***

distributed memory machine see ***DDLY, High Performance Fortran (HPF), PASSION***

distributed view see ***Application Program Interface, Jovian***

distribution

The term distribution determines in which segment the record of a file resides and where in that segment. It is equivalent to a one-to-one mapping from file record number to a pair containing segment number and segment record number [73]. See also ***PPFS***.

distribution file pointer thread (DFP) see ***SPIFFI***

distributed computing

Distributed computing is a process whereby a set of computers connected by a network is used collectively to solve a single large program [55]. **Message passing** is used as a form of interprocess communication.

distribution pattern see ***design of parallel I/O software***

DRA see ***Disk Resistent Arrays***

E

ELFS see ***ExtensibLe File Systems***

EPVM (E persistent Virtual Machine) see ***programming language E***

ESM (EXODUS storage manager) see ***programming language E, SHORE***

executor see ***PARTI***

EXODUS

EXODUS an ***object-oriented database*** effort and serves as the basis for ***SHORE***. EXODUS provides a **client-server** architecture and supports multiple servers and transactions [21]. The ***programming language E***, a variant of C++, is included in order to support a convenient creation and manipulation of persistent data structures. Although EXODUS has good features such as transactions, performance and robustness, there are some important drawbacks [21]: storage objects are untyped arrays of bytes, no type information is stored, it is a **client-server** architecture, it lacks of support for access control, and existing applications built around UNIX files cannot easily use EXODUS.

EXODUS storage manager (ESM) see ***SHORE, programming language E***

Explicit Communication see ***PASSION***

Express

Express is a toolkit that allows to individually address various aspects of concurrent computation [55]. Furthermore, it includes a set of libraries for communication, I/O and parallel graphics.

extended memory see ***design of parallel I/O software***

Extended TPM (ETPM) see ***Two-Phase Method (TPM)***

ExtensibLe File Systems (ELFS)

abstract:

ELFS is based on an object-oriented appraoch, i.e. files should be treated as typed objects. Ease-of-use can be implemented in a way that a user is allowed to manipulate data items in a manner that is more natural than current file access methods available [79]. For instance, a 2D matrix interface can be accessed in terms of rows, columns or blocks. In particular, the user can express requests in a manner that matches the semantic model of data [79], and does not have to take care of the physical storage of data, i.e. in the object-oriented approach the implementation details are hidden. Ease of development is supported by encapsulation and inheritance as well as code reuse, extensibility and modularity.

aims:

high performance, ease of development and maintenance, and ease-of-use; four key ideas [79]:

1. design the user interface to support ease-of-use
2. improve performance by matching the file structure to the access patterns of the application and the type of data

3. selectively employ advanced I/O access techniques (***prefetching***, ***caching***, multiple I/O threads)
4. encapsulate the implementation details

implementation platform:

Intel iPSC/hypercube + CFS

application:

radio astronomy applications [77], [78]

people:

John F. Karpovich, Andrew S. Grimshaw, James F. French

{jfk3w, grimshaw, french}@virginia.edu

institution:

Department of Computer Science, University of Virginia, Charlottesville, VA, USA

http://www.cs.viginia.edu

key words:

file system, object-orientation

example:

The following code fragments from [79] both read a specific column of integers from a matrix. Program1 uses traditional UNIX style file operations whereas Program2 applies an auxiliary two dimensional data class DD_intarray.

Program1

```
int i,fd;
int m[maxRows][maxCols];

fd = open (filename);
for (i = 0; i < numRows; i++)
{
  //calculate file position for row i, column x
  seek (fd, position);
  read (fd, numBytes);

  //convert data if necessary
  m[i][x] = val;
}
...
close (fd);
```

Program2

```
TwoDMatrixFile f;

f.openFormatted (filename);
maxRows = f.numRows ();
maxCols = f.numColumns ();

DD_intarray m(maxRows, maxCols);
m.column (2) = f.readColumn (2);
...
f.close ();
```

external computing see ***design of parallel I/O software***

F

F77 (Fortran 77) see ***Fortran D, MPI, PASSION***

FD (File Domain) see ***Two-Phase Method***

FDAT (File Domain Access Table) see ***Two-Phase Method***

FE (front end computer) see ***SIMD***

File Access Descriptor (FAD) see ***Two-Phase Method***

File Domain (FD) see ***Two-Phase Method***

File Domain Access Table (FDAT) see ***Two-Phase Method***

file level parallelism

A conventional file system is implemented on each of the processing nodes that have disks, and a central controller is added which controls a transparent striping scheme over all the individual file systems. The name file level parallelism stems from the fact that each file is explicitly divided across the individual file systems. Moreover, it is difficult to avoid arbitrating I/O requests via the controller (**bottleneck**). ***HiDIOS*** has introduced a **disk level parallelism** (parallel files vs. parallel disks) [148].

file migration

The amount of data gets larger and larger, hence, storing this data on a magnetic disk is not always feasible. Instead, tertiary storage devices such as tapes and optical disks are used. Although the costs per megabyte of storage are lowered, they have longer access times than magnetic disks. A solution to this situation is to use file migration systems that are used by large computer installations to store more data than that which would fit on magnetic disks [105].

file server see ***RAID-I, RAID-II***

file services see ***ParFiSys***

FLEET

FLEET is a FiLEsystem Experimentation Testbed at Dartmouth College, USA, for experimentation with new concepts in ***parallel file system***s. See also ***disk-directed I/O***.

flush ahead see ***read ahead***

Fortran 90 see ***CHAOS, Global Arrays (GA), High Performance Fortran (HPF), MPI-2, Pablo***

Fortran D

abstract:

Fortran D is a version of Fortran that provides data decomposition specifications for two levels of parallelism (how should arrays be aligned with respect to each other, and how should arrays be distributed onto the parallel machine) [91] (See also ***High Performance Fortran (HPF)***.). Furthermore, a Fortran D compilation system translates a Fortran D program into a **Fortran 77** ***SPMD*** node program. A consequence can be a reduction or hiding of communication overhead, exploited parallelism or the reduction of memory requirements.

aims:

Fortran D is designed to provide a simple, efficient machine-independent ***data parallel*** programming model [70].

people:

Ken Kennedy, Seema Hiranandani, Chan-Wen Tseng

{ken, seema}@cs.rice.edu

tseng@stanford.edu

institution:

Center for Research and Parallel Computing, Rice University, Houston, TX 77251-1892, USA

http://www.crpc.rice.edu/fortran-tools/fortran-tools.html

Fortran M (FM) see ***task-parallel program***

front end computer (FE) see ***SIMD***

Fujitsu AP1000

The AP1000 is an experimental large-scale ***MIMD*** parallel computer with configurations range from 64 to 1024 processors connected by three separate high-bandwidth communication networks. There is no **shared memory**, and the processors are typically controlled by a host like the **SPARC** Server. A processor is a **SPARC** 25MHz, 16MB RAM processor. Programs are written in C or Fortran. ***HiDIOS*** is a ***parallel file system*** implemented on the AP1000.

G

GA see ***Global Arrays***

GAF (Global Array File) see ***Global Placement Model (GPM)***

Galley

abstract:

Galley is a ***parallel file system*** intended to meet the needs of parallel scientific applications (see aims). It is based on a three-dimensional structuring of files [115].

Furthermore, it is supposed to be capable of providing high performance I/O.

aims:

It was believed that parallel scientific applications would access large files in large consecutive chunks, but results have shown that many applications make many small, regular, but non-consecutive requests to the file system. Galley is designed to satisfy such applications. The goals are [114]:

- efficiently handle a variety of access sizes and patterns
- allow applications to explicitly control parallelism in file access
- be flexible enough to support a variety of interfaces and policies, implemented in libraries
- allow easy and efficient implementations of libraries
- scale to many compute and **I/O processor**s
- minimize memory and performance overhead

implementation platform:

IBM SP2, it also runs on most UNIX clusters

data access strategies:

Galley allows different kinds of operations: **meta-data** operations like file, fork and data operations, and data operations like simple, batch and non-blocking operations.

portability:

ViC* is planed to be implemented on top of Galley

MPI, ***PVM*** or ***p4*** can be used as communication software

application:

[115] presents how a Flexible Image Transport System can be implemented with Galley.

people:

Nils Nieuwejaar, David Kotz

`{nils, dfk}@cs.dartmouth.edu`

institution:

Department of Computer Science, Dartmouth College, Hanover, NH 03755-3510, USA

`http://www.cs.darmouth.edu/ nils/galley.html`

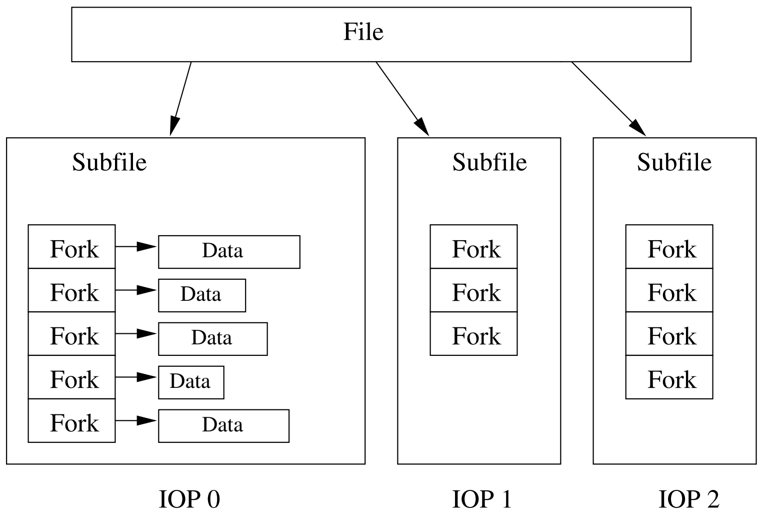


Figure 2.5: 3-D Structure of Files in Galley

key words:

file system, **client-server**, 3-D file structuring

details:

1 File Structure

Galley is not based on the traditional UNIX-like file system interface, but Galley provides the application with the ability to control the **declustering** of a file [114]. What is more, the disk that should be accessed can also be indicated explicitly. This is achieved by composing a file of one or more subfiles, each residing on a single disk and each being directly addressed by the application. As a result, data ***distribution*** and the degree of parallelism can be controlled. A subfile is decomposed and consists of one or more independent forks, which is a linear, addressable sequence of bytes. The number of subfiles in a file is fixed whereas the amount of forks is variable. The whole file structure is depicted in Figure 2.5. This concept can be used for storing related information - that is available independently - logically together.

2 System Structure

Galley is based on the **client-server** model where a client is a user application linked with the Galley run-time library running one a **compute processor (CP)** [114]. In particular, the run-time library receives requests from the application, translates them into lower-level requests and passes them to the servers on **I/O processors (IOP)**. It is the task of the library to handle the data transfer between **compute** and **I/O processor**s. Since Galley does not impose communication requirements on a user's application, any communication software like ***MPI***, ***PVM*** or ***p4*** can be used.

Each **IOP** is composed of several units and has one thread to handle incoming I/O requests for each **compute processor**. Multithreading allows to service requests simultaneously. An **IOP** receives a request from a **CP**. Thereafter a **CP Thread** interprets the requests, forwards them on to the appropriate worker thread and handles the transfer of data between **IOP** and **CP**. As in ***Vesta***, **meta-data** is distributed across all **IOP**s. The data transfer also includes a **cache manager** which maintains a separate list of requested disk blocks for each thread, and a **disk manager** which maintains a list of blocks that are scheduled to be read or written.

General Purpose MIMD machine see ***GPMIMD***
GFP (global file pointer thread) see ***SPIFFI***
GFP thread see ***SPIFFI***

Global Arrays (GA)

abstract:

Global Arrays are supposed to combine features of **message passing** and **shared memory**, leading to both simple coding and efficient execution for a class of applications that appears to be fairly common [112]. Global arrays are also regarded as "A Portable 'Shared Memory' Programming Model for Distributed Memory Computers". GA also support the NUMA (Non-Uniform Memory Access) **shared memory** paradigm. What is more, two versions of GA were implemented: a fully distributed one and a mirrored one. See also ***Disk Resistent Arrays(DRA)***.

aims:

GA provides a portable interface where each process in a ***MIMD*** parallel program can independently, asynchronously and efficiently access logical blocks of physically distributed matrices, with no need for explicit cooperation by other processes.

implementation platform:

Libraries and tools have been implemented on the ***Intel Touchstone Delta*** and ***Intel Paragon***, **IBM SP1**, ***Kendal Square KSR-2*** and UNIX workstations. In general, GA support distributed and **shared memory machines**.

data access strategies:

- operations have implied synchronization; an asynchronous mode is available, too.
- true ***MIMD*** style operation invocation

portability:

Basic functionalities may be expressed as single statements using **Fortran-90** array notation. Operations can be used for Fortran and C.

interfaces to: ScaLAPACK, PeIGS parallel eigensolver library, SUMMAparallel matrix multiplication, ***MPI, PVM***

application:

large chemistry applications for the HPCCI project and EMSL [112];

Some experiments with NWChem, a complex chemistry package on top of GA, were performed.

people:

Jaroslav Nieplocha, Robert J. Harrison, Richard J. Littlefield

{j_nieplocha, rj_harrison, rj_littelfield}@pnl.gov

institution:

Pacific Northwest Laboratory, P.O. BOX 999, Richland, WA 99352, USA

http://www.emsl.pnl.gov:2080/docs/global/ga.html

key words:

MIMD, **distributed memory**, **shared memory**, networked workstation clusters

example:

The example uses a FORTRAN interface to create an n x m double precision array, blocked in at least 10x5 chunks [112].

```
integer g_a, n, m, ilo, ihi, jlo, jhi, ldim
double precision local (1:ldim, *)
```

```
c
call ga_create (MT_DBL, n, m, 'A', 10, 5, g_a)
call ga_zero (g_a)
call ga_put (g_a, ilo, ihi, jhi, local, ldim)
```

details:

The current GA programming model can be characterized as follows [112]:

- ***MIMD*** parallelism is provided using a multi-process application, in which all non-GA data, file descriptors, and so on are replicated or unique to each process.
- Processes can communicate with each other by creating and accessing GA distributed matrices.
- Matrices are physically distributed blockwise, either regularly or as the Cartesian product of irregular ***distribution***s (see ***irregular problems***) on each axis.
- Each process can independently and asynchronously access any two-dimensional patch of a GA distributed matrix, without requiring cooperation by the application code in any other process.
- Each process is assumed to have fast access to some portion of each distributed matrix, and slower access to the remainder. These speed differences define the data as being 'local' or 'remote'.
- Each process can determine which portion of each distributed matrix is stored 'locally'. Every element of a distributed matrix is guaranteed to be 'locally' to exactly one process.

In comparison to common models, GA are different since they allow **task-parallel** access to distributed matrices. Furthermore, GA support three distinctive environments [112]:

- **distributed memory**, **message passing** parallel computers with interrupt-driven communication (Intel Gamma, ***Intel Touchstone Delta*** and ***Intel Paragon***, **IBM SP1**)

- networked workstation clusters with simple **message passing**
- **shared memory** parallel computers (***KSR-2***, **SGI**)

Global Array File (GAF) see ***Global Placement Model (GPM)***
global file pointer thread (GFP) see ***SPIFFI***

Global Placement Model (GPM)

In ***PASSION*** there are two models for storing and accessing data: the **Local Placement Model (LPM)** and the Global Placement Model (GPM) [14]. For many applications in supercomputing main memory is too small, therefore, main parts of the available data are stored in an array on disk. The entire array is stored in a single file, and each processor can directly access any portion of the file [31]. In a GPM a global data array is stored in a single file called **Global Array File (GAF)** [33]. The file is only logically divided into local arrays, which saves the initial local file creation phase in the **LPM**. However, each processors' data may not be stored contiguously, resulting in multiple requests and high I/O latency time.

global view see ***Application Program Interfac, Jovian, VIP-FS***

GPMIMD (General Purpose MIMD)

A general purpose multiprocessor I/O system has to pay attention to a wide range of applications that consist of three main types: normal UNIX users, databases and scientific applications [23]. Database applications are characterized by a multiuser environment with much random and small file access whereas scientific applications support just a single user having a large amount of sequential accesses to a few files.

The main components are **processing nodes (PN)**, network, input/output nodes (**ION**) and disk devices. In order to describe a system, four parameters can be used [23]: number of **I/O node**s, number of controllers, number of disks per controller, and degree of synchronization across disks of a controller. Additionally, another two concepts must be considered: file ***clustering*** and **file striping**. A declustered file is distributed across a number of disks

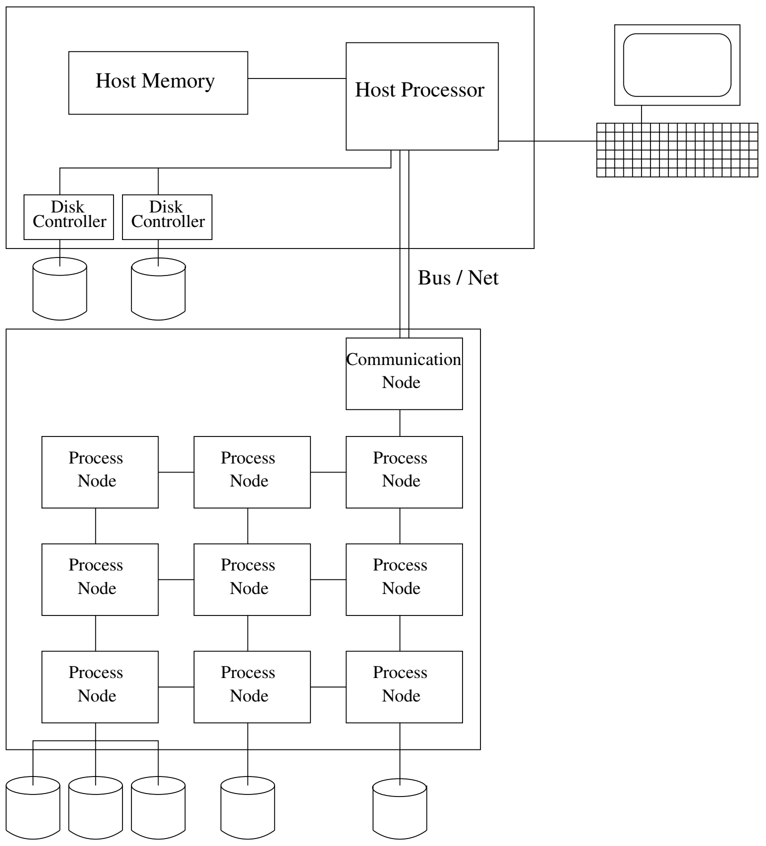


Figure 2.6: GPMIMD Architecture

such that different blocks of the same file can be accessed in parallel from different disks. In a stripped file a block can be read from several disks simultaneously.

The GPMIMD machine is a multiprocessor system with integrated hardware and software [25]. The communication between the GPMIMD and the host is accomplished using the communication channels of a communication node of the GPMIMD machine (see Figure 2.6). The front-end computer (on which the system software is based) is a gateway to the GPMIMD system which provides software development, debugging and control tools. The front-end computer can be any computer system and has the following three primary functions in the GPMIMD system [25]:

- It provides a development and debugging environment for applications.
- It runs applications and transmits instructions to the parallel processing units.

- It provides maintenance and operation utilities for controlling the GPMIMD machine.

Grand Challenge Applications

Massively parallel processors (MPPs) are more and more used in order to solve Grand Challenge Applications which require much computational effort [31]. They cover fields like physics, chemistry, biology, medicine, engineering and other sciences. Furthermore, they are extremely complex, require many Teraflops of communication power and deal with large quantities of data. Although supercomputers (see ***supercomputing applications***) have large main memories, the memories are not sufficiently large to hold the amount of data required for Grand Challenge Applications. High performance I/O is necessary if a degrade of the entire performance of the whole program has to be avoided. Large scale applications often use the ***Single Program Multiple Data (SPMD)*** programming paradigm for ***MIMD*** machines [12]. Parallelism is exploited by decomposing of the data domain.

H

Hamming code see ***RAID***

HFS see ***Hurricane File System (HFS)***

HiDIOS (High performance Distributed Input Output System)

abstract:

HiDIOS (part of the ***CAP Research Program***) is a ***parallel file system*** for the ***Fujitsu AP1000*** multicomputer. What is more, HiDIOS is a product of the ACSys ***PIOUS*** project. HiDIOS uses a **disk level parallelism** (instead of the ***file level parallelism***) [148] where a parallel disk driver is used which combines the physically separate disks into a single large parallel disk by stripping data cyclically across the disks. Even the file system code is written with respect to the assumption of a single large, fast disk.

aims:

- high speed, robustness, usability

- efficient use of a large number of disks using a fast network
- maximize disk throughput for large transfers
- present a single coherent file system image
- portability, **scalability**
- low impact on processor and memory resources

implementation platform:

Fujitsu AP1000 multicomputer

data access strategies:

In HiDIOS the scheduling problem is solved by threads, i.e. each I/O request spawns a new thread, and threads also control the queuing of disk operations and the allocation of memory resources.

portability:

AP shell can be invoked from a front end host of an ***Fujitsu AP1000*** that supports ***HiDIOS***

related work:

PIOUS, ***MPI***

people:

Andrew Tridgell, David Walsh

{`andrew.tridgell, david.walsh`}`@cs.anu.edu.au`

institution:

Australian National University

`http://cafe.anu.edu.au/cap/projects/filesys/`

key words:

file system, **distributed memory**, **disk level parallelism**

details:

Requests are placed in request queues, which are thereafter processed by a number of independent threads [148]. After request processing the manager can return and, hence, can receive further requests while previous ones may be blocked waiting for disk I/O. The **meta-data** system makes it possible to immediately service **meta-data** manipulation (such as file

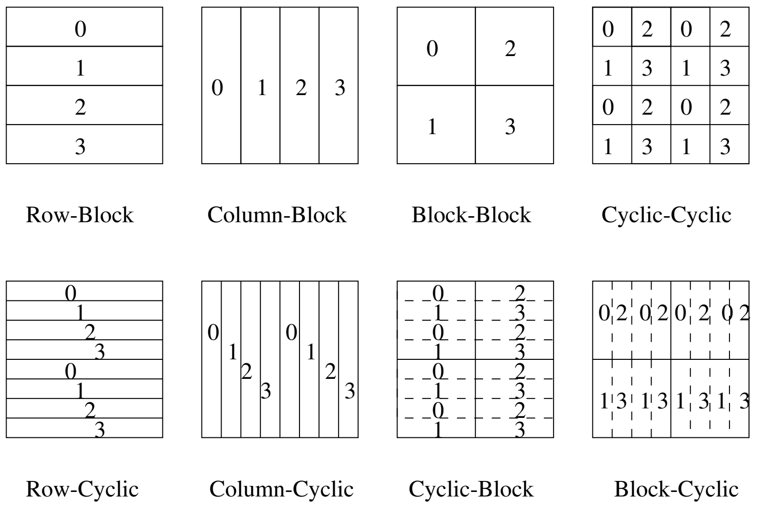


Figure 2.7: Data Distribution in Fortran 90D/HPF

creation, renaming) without disk I/O.

Hierarchical Clustering see ***Hurricane File System (HFS)***

HPF (High Performance Fortran)

High Performance Fortran is an extension to **Fortran 90** with special features to specify data ***distribution***, alignment or ***data parallel*** execution [141], and it is syntactically similar to ***Fortran D*** [12]. HPF was designed to provide language support for machines like ***SIMD***, ***MIMD*** or vector machines. Moreover, it provides directives like `ALIGN` and `DISTRIBUTE` for distributing arrays among processors of **distributed memory machines**. Here an array can either be distributed in a block or cyclic fashion. Figure 2.7 depicts eight ***distribution*** styles supported by HPF. `DISTRIBUTE` specifies which parts of the array are mapped to each processor, i.e. each processor has a local array associated with it.

A sample program in HPF looks as follows:

```
REAL A(200), B(200), C(200,200)
$CHPF PROCESSORS PRC(4)
$CHPF TEMPLATE DUMMY (200)
$CHPF DISTRIBUTE DUMMY (BLOCK)
```

```
$CHPF ALIGN (i) with DUMMY(i):: A, B
$CHPF ALIGN (*,i) with DUMMY(i):: C\\
```

HPF is also supposed to make programs independent of single machine architectures [44]. Although HPF can reduce communication cost and, hence, increase the performance, this is only true for regular but not for ***irregular problems***.

HPSS (High-Performance Storage System) see ***Network-Attached Peripherals (NAP), Scalable I/O Facility***

Hurricane File System (HFS)

abstract:

The Hurricane File System is developed for large-scale **shared memory** multicomputers. Since application I/O requirements for large-scale multiprocessors are not yet well understood, the file system must be flexible so that it can be extended to support new ***I/O interfaces*** [95].

aims:

The mail goals are **scalability**, flexibility and maximize I/O performance for a large set of parallel applications. There are three important features of HFS:

- Flexibility: HFS can support a wide range of file structures and file system policies.
- Customizability: The structure of a file and the policies invoked by the file system can be controlled by the application.
- Efficiency: little I/O and CPU overhead

implementation platform:

HFS was ported to: **IBM R6000/350**, Sun 4/40 with SunOS Version 4.1.1, **SGI** Iris 4d/240S IRIX System V release 3.3.1 with AIX Version 3.2

data access strategies:

storage objects

related work:

CFS, ***Vesta***, ***OSF/1***, ***nCUBE, Bridge file system, RAMA file system***

people:

Orran Krieger, Michael Stumm, Karen Reid, Ron Unrau

{okrieg, stumm, reid, unrau}@eecg.toronto.edu

institution:

Department of Electrical and Computer Engineering, University of Toronto, Canada, M5S 1A4

http://www.eecg.toronto.edu/parallel/hurricane.html

key words:

shared memory, file system, hierarchical ***clustering***, **storage object**

example:

The code fragments from [96] illustrate the funtions `salloc` and `sfree`, and a read function implemented using **ASI**.

```
void *salloc (FILE *iop, int *lenptr)
{
  void *ptr;
  AquireLock (iop->lock);
  ptr = iop->bufptr;
  if (iop->bufcnt >= *lenptr)
  {
    iop->refcnt++;
    iop->bufcnt -= *lenptr;
    iop->bufptr += *lenptr;
  }
  else
    ptr = iop->u_salloc (iop, lenptr);
  ReleaseLock (iop->lock);
  return ptr;
}


void sfree (FILE *iop, void *ptr)
{
  int rc = 0;
  AcquireLock (iop-lock);
  if ((ptr <= iop->bufptr) && (ptr >= iop->bufbase)
    iop->refcnt--;
  else
    rc = iop->u_sfree (iop, ptr);
  ReleaseLock (iop->lock);
```

```
    return rc;
  }


  int read (int fd, char *buf, int length)
  {
    int error;
    FILE *stream = streamptr (fd);
    if (ptr = Salloc (stream, SA_READ, &length))
    {
      bcopy (ptr, buf, length);
      if (!error = sfree (stream)))
        return length;
    }
    else
      error = length;
    RETURN_ERR (error);
  }
```

details:

Distributing the disks across the system has the advantage that some disks can be made more local to a processor which reduces access time and costs. So the file system must try to manage locality so that a processor's I/O requests are primarily directed to nearby devices.

1 Hierarchical Clustering

The basic unit is the cluster, which provides the full functionality of a small-scale operating system [95] (see Figure 2.8). On a large system, multiple clusters are instantiated such that each cluster manages a unique group of 'neighboring' processors where 'neighboring' implies that memory access within a cluster is less expensive than accesses to another cluster. Furthermore, all system services are replicated to each cluster. Clusters cooperate and communicate in a loosely coupled fashion. For larger systems extra levels of clusters can be added hierarchically (super clusters, super super clusters). If possible, requests are handled within a cluster, but non-independent requests are resolved by servers which are located at the higher levels in the cluster hierarchy. The costs of non-independent requests depend on the degree of sharing those requests.

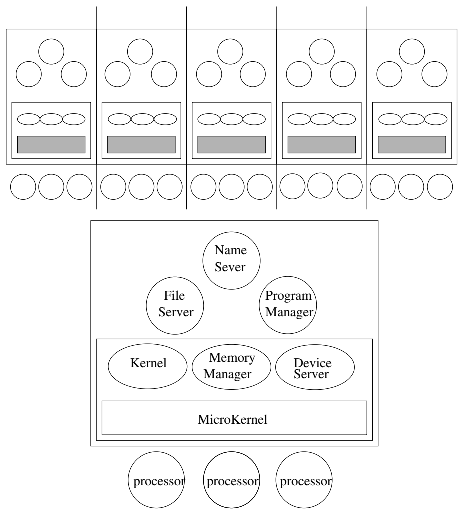


Figure 2.8: Multiple Hurricane Clusters

2 The File System Architecture

HFS is a part of the ***Hurricane operating system*** (see Figure 2.9). The file system consists of three user level system servers: the **Name Server, Open File Server (OFS)** and **Block File Server (BFS)**.

- The **Name Sever** manages the name space and is responsible for authenticating requests to open files.
- The **OFS** maintains the file system state kept for each open file.
- The **BFS** controls the system disks, is responsible for determining to which disk an operation is destined and directs the operation to the corresponding device driver.
- **Dirty Harry (DH)** collects dirty pages from the memory manager and makes requests to the **BFS** to write the pages to disk.
- The ***Alloc Stream Facility (ASF)*** is a user level library. It maps files into the application's address space and translates read and write operations into accesses to mapped regions.

Each of those file system servers maintains a different state. Whereas the **Name Server** maintains a logical directory state [95] (e.g. access permission and directory size) and di-

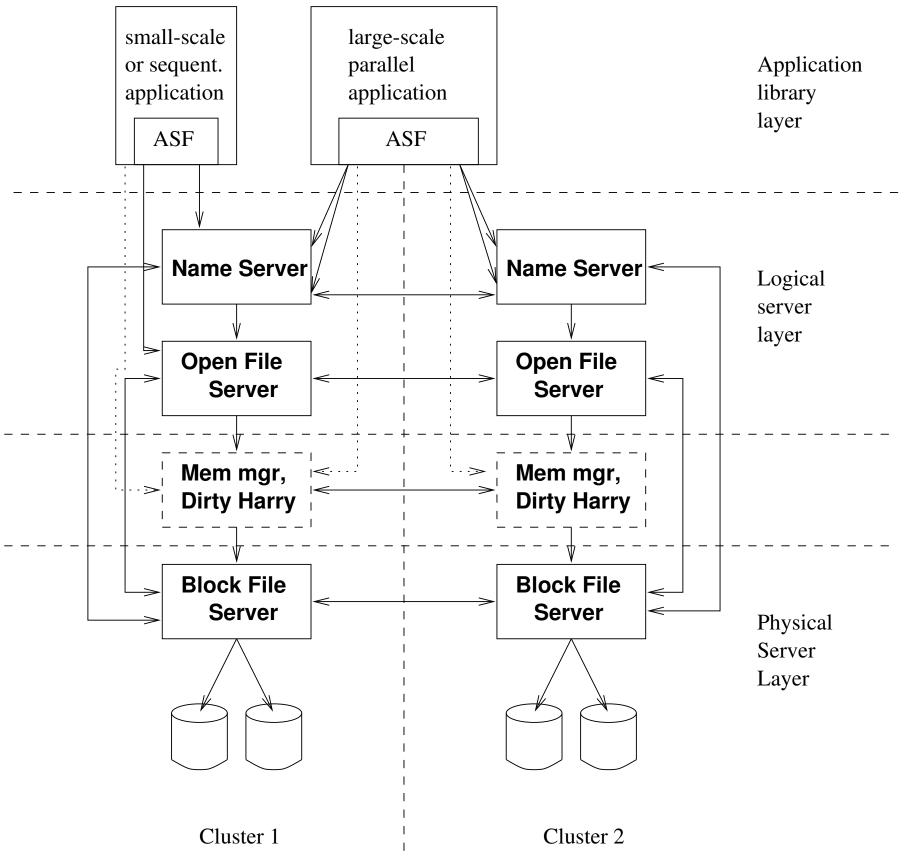


Figure 2.9: The Hurricane File System

rectory contents, the **OFS** maintains logical file information (length, access permission, ...) and the per-open instance state. Finally, the **BFS** maintains the block map for each file. Obviously, these states are different from each other and independent, consequently, there is no need for different servers within a cluster to communicate in order to keep the state consistent [95].

Example: A file is opened and ***ASF*** sends the 'open' to the **Name Server** which translates the file name into its token (a number which identifies the file). After checking whether access is allowed the request is forwarded to the **BFS**. The **BFS** uses the token to get information about the file state and forwards this information to the **OFS**. The **OFS** records the file state and the file token and returns a file handle to the application.

3 Storage Objects

For a file system it is important to be easily extensible to support new policies, so HFS uses an object oriented building-block approach [93]. Files are implemented by combining together a number of simple building blocks called **storage object**s. Each **storage object** defines a part of a file's structure and encapsulates **meta-data** and member functions that operate on member data. Since HFS allows applications control over the **storage object**s, the application can define the internal structure of files as well as the policies. In contrast to most existing file systems, HFS supports an unlimited set of file structures and policies. Every **storage object** is made up of three components: one for the **BFS**, one for the **OFS** or the **Name Server**, and one for the application library (***ASF***). For code reuse, **storage object**s can be derived from other **storage object**s.

A **storage object** can either be persistent or transitory, depending whether the **storage object** is stored on disk (persistent) or if it exists just when a file is actively accessed (transitory).

- Transitory **storage object**s contain all file system functionalities that are not specific to the file structure and, hence, do not have to be fixed at file creation time. Examples are latency-hiding policies, compression policies, advisory locking policies etc.

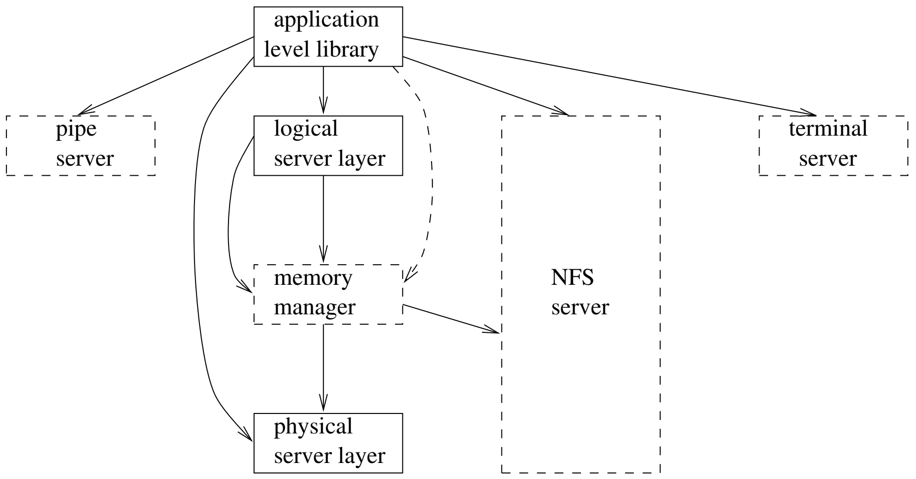


Figure 2.10: The HFS architecture

- Persistent **storage object**s define the structure of a file, store file data and file system states on disk.

4 File system layers

HFS consists of the following three logical layers: application-level library, physical server layer and logical server layer (see Figure 2.10).

The application-level library is the library for all other **I/O server**s provided by the ***Hurricane operating system***. It has the aim to service as much application requests as possible within the application-level. The physical layer directs requests for file data to the disks and maintains the mapping between file blocks. Furthermore, it is responsible for file structure and policies. All **Physical layer Storage Objects (PSO)** are persistent. The logical server layer implements all file system functionalities that do not have to be implemented at the other layers.

Hurricane operating system

It is structured according to the **Hierarchical Clustering** in ***HFS***, i.e. the file system on each cluster provides the functionality of a full, small scale file system. The file system state is cached whenever possible to minimize cross-cluster communication. The state is kept consistent by per-cluster-server communication. This also contributes to the ***distribution*** of disk I/O.

Hurricane is a micro-kernel and single storage operating system that supports mapped file I/O. A mapped file system allows that the application can map regions of a file into its address space and access the file by referencing memory in mapped regions. Moreover, main memory can be used as a cache of the file system.

Another feature of Hurricane is a facility called **Local Server Invocations (LSI)** that allows a fast, local, cross-address space invocation of server code and data, and results in new workers being created in the server address space. **LSI** also simplifies **deadlock** avoidance [95].

I

i-node see ***Portable Parallel File System (PPFS), Vesta***
I/O daemon see ***concurrency algorithms, parallel file system, PIOUS, S-2PL***
I/O Device Drive Layer (IDD) see ***Virtual Parallel File System***

I/O interfaces

[93] describes three main classes of I/O interfaces

- read/write system-call interfaces: The best known interface of this type is the UNIX one (see ***UNIX I/O***). System calls like `open, read, write, lseek, close, fcntl` and `ioctl` are used.
- mapped file system-call interface: It is supported by most modern operating systems. Briefly, a contiguous memory region of an application address space can be mapped to a contiguous file region on secondary storage. Furthermore, a uniform interface for all I/O operations is guaranteed, i.e. the same I/O operations can be used for either files,

terminals or network connections.

- application-level interfaces: An example is the standard I/O library of the C programming language. The functions provided by `stdio` correspond to the ones of the ***UNIX I/O*** interface. See also ***Alloc Stream Facility***.

In addition to the **sequential I/O interface**s mentioned above, there are also several **parallel I/O interface**s. A 'good' **parallel I/O interface** should have following properties [94]:

- flexibility: simple for novice programmers, but still satisfying the performance requirements for expert programmers. Additionally, an application should be free to choose the amount of policy related information it specifies to the system.
- incremental control: A programmer should be able to write a functionally correct program and then incrementally optimize its I/O performance. Additional information should be provided in order to get better performance.
- dynamic policy choice: Applications should be allowed to change dynamically the policies used.
- generality: A policy should refer to both explicit and implicit I/O.
- portability: An interface should be applicable to many different parallel systems (distributed, **shared memory**).
- low overhead
- concurrency support: Well defined semantics are required when multiple threads access the same file.
- compatibility

I/O node (ION) see ***ADOPT, CCFS, disk-directed I/O, Intel iPSC/860 hypercube, Intel Touchstone Delta, Panda, ParFiSys (Parallel File System), PIOUS, PPFS, S-2PL, Vesta***

I/O problem

The I/O problem (also referred to as the **I/O bottleneck problem** in [16]) stems from that fact that the processor technology is increasing rapidly, but the performance and the access time of secondary storage devices such as disks and floppy disk drives have not improved to the same extend [50]. Disk seek times are still low, and I/O becomes an important **bottleneck**. The gap between processors and I/O systems is increased immensely, which is especially obvious and tedious in multiprocessor systems. However, the I/O subsystem performance can be increased by the usage of several disks in parallel. As for the ***Intel Paragon*** XP/S, ***RAID***s are supported.

I/O processor see ***Galley***
I/O Requirement Graph (IORG) see ***ViPIOS***
I/O server see ***ANL, HFS, ParFiSys, PPFS***
IBL (Infinite Block Lookahead) see ***read ahead***
IBM 9570 RAID see ***ANL (Argonne National Laboratory)***
IBM AIX Parallel I/O File System see ***PIOFS***
IBM R6000/350 see ***HFS***
IBM RS/6000 see ***ANL (Argonne National Laboratory), ChemIO, SPFS, Vesta***
IBM SP see ***ADIO, ROMIO***
IBM SP1 see ***ANL (Argonne National Laboratory), Global Array (GA), Multipol, Vesta, ViPIOS***

IBM SP2

SP2 is a homogeneous parallel system since the nodes are binary compatible, but it is very heterogeneous in the point of view of the availability of local disks, direct network connections and ***distribution*** of physical memory [127]. See also Appendix: Parallel Computer Architecture.

IDD (I/O Device Drive Layer) see ***Virtual Parallel File System***

in-core communication

In-core communication can be divided into two types: **demand-driven** and **producer-**

driven [14].

- **demand-driven**: The communication is performed when a processor requires off-processor data during the computation of the **ICLA** (see ***PASSION***). A node sends a request to another node to get data.

- **producer-driven**: When a node computes on an **ICLA** and can determine that a part of this **ICLA** will be required by another node later on, this node sends that data while it is in its present memory. The producer decides when to send the data. This method saves extra disk access, but it requires knowledge of the data dependencies so that the processor can know beforehand what to send.

In-core Communication Method see ***PASSION***

In-core Local Array (ICLA) see ***PASSION, data sieving***

independent disk addressing see ***parallel file system***

Infinite Block Lookahead (IBL) see ***read ahead***

information disk see ***coding techniques***

inspector see ***PARTI***

Intel forced message types see ***PARTI***

Intel iPSC/860 hypercube

The Intel iPSC/860 is a **distributed memory**, **message passing** ***MIMD*** machine, where the **compute node**s are based on Intel i860 processors that are connected by a hypercube network [90]. **I/O node**s are connected to a single **compute node** and handle I/O. What is more, **I/O node**s are based on the Intel i386 processor. See also ***PARTI***, Appendix: Parallel Computer Architecture.

Intel Paragon

The Intel Paragon (also referred to as Intel Paragon XP/S) multicomputer has its own operating system ***OSF/1*** and a special file system called ***PFS (Parallel File System)***. The Intel Paragon is supposed to address ***Grand Challenge Applications*** [75]. In particular, it is a **distributed memory** multicomputer based on Intel's teraFLOPS architecture.

More than a thousand heterogeneous nodes (based on the Intel i860 XP processors) can be connected in a two-dimensional rectangular mesh. Furthermore, these nodes communicate via **message passing** over a high-speed internal interconnect network. A ***MIMD*** architecture supports different programming styles including ***SPMD*** and ***SIMD***. However, it does not have **shared memory** [54]. ***SPIFFI*** is a scalable ***parallel file system*** for the Intel Paragon.

Intel Touchstone Delta

The Intel Touchstone Delta System is a **message passing** multicomputer consisting of processing nodes that communicate across the two dimensional mesh interconnecting network. It uses Intel i860 processors as the core of communication nodes. In addition, the Delta has 32 Intel 80386 processors as the core of the **I/O node**s where each **I/O node** has 8 Mbytes memory that serves as I/O cache [15]. Furthermore, other processor nodes such as service nodes or ethernet nodes are used.

inter-communicator see ***MPI***

Interface layer see ***Virtual Parallel File System***

Internetworking see ***operating system components***

Interprocedural Partitial Redundancy Elimination algorithm (IPRE) see ***PRE***

intra-communicator see ***MPI***

Intranetworking see ***operating system components***

ION see ***ADOPT, CCFS, disk-directed I/O, Intel iPSC/860 hypercube, Intel Touchstone Delta, Panda, ParFiSys (Parallel File System), PIOUS, PPFS, S-2PL, Vesta***

IORG (I/O Requirement Graph) see ***ViPIOS***

IOP (I/O processor) see ***Galley***

IPRE see ***Partial Redundancy Elimination (PRE)***

irregular (unstructured) problems

[152] introduces three different kinds of irregularity:

- irregular control structures: These are conditional statements making it inefficient to run on synchronous programming models.
- irregular data structures: unbalanced trees or graphs
- irregular communication patterns: leads to non-determinism

*1 Irregular Problems in **PASSION***

Basically, in irregular problems data access patterns cannot be predicted until runtime [32]. Consequently, optimizations carried out at compile-time are limited. However, at run-time data access patterns of nested loops are usually known before entering the loop-nest, which makes it possible to utilize various preprocessing strategies. The following ***HPF***-example illustrates a typical irregular loop, where `array1` is known only at run-time.

```
real x(n_node), y (n_node)
integer array1(n_edge, 2)

do i = 1, n_step
   do j = 1, n_edge
      x(array1(i, 1)) = x(array(i, 1)) + y(array1(i,2)
      x(array1(i, 2)) = x(array(i, 2)) + y(array1(i,1)
   end do
end do
```

PASSION contains run-time routines to solve **out-of-core** irregular problems on **distributed memory** machines. The **OOC** computation is executed in three stages: data and/or indirection array partitioning, pre-processing of the indirection array and actual computation.

[17] also deals with irregular problems, but bases its work on ***PASSION***. In particular, [17] presents the design of various steps, runtime system and compiler transformations to support irregular **out-of-core** problems. The strategy applied uses three code phases (similar to ***CHAOS***) called **work distributor**, **inspector** and **executor**. The goal of the optimization is to minimize communication.

Two kinds of **OOC** problems are considered in [17]: the first one is based on the assumption that data can fit into the system's memory whereas data structures describing interactions are **OOC**. This class can be referred to as **Data Arrays are In-Core and Indirection Arrays are Out-of-core (DAI/IAO)**. [17] states seven steps in which the **DAI/IAO** computation is performed:

1. Default Initial and Work Distribution
2. Partition Data Arrays
3. Redistribute Data Arrays
4. Compute new iteration tiles
5. Redistribute local files with indirection arrays
6. Out-of-Core Inspector
7. Out-of-Core Executor

Some experiments have been performed using a program which includes the ***data parallel*** loops taken from the unstructured 3-d Euler solver. The experiments were carried out on an ***Intel Paragon***.

The second class of problems considered in [17] is a generalized version of the **OOC** problem, i.e. both the data arrays and the interaction/indirection arrays are **OOC**. Here an update value of a node may also require disk I/O. Hence, the maximum number of updates should be preferred in one update I/O operation. The approach is based on the **iteration loop transformation**. The steps 1 to 5 are the same, only 6 and 7 from above have to be changed and applied to the new problem.

2 Irregular Problems in CHAOS

[120] introduces four basic steps which are necessary to solve irregular problems:

1. data partitioning
2. partitioning computational work
3. software ***caching*** methods to reduce communication volume

4. communication vectorization to reduce communication startup costs

The languages ***Vienna Fortran*** and ***Fortran D*** also allow the user to specify a customized ***distribution*** function which is in contrast to the standard data ***distribution*** like block or cyclic.

Applications using the ***CHAOS***-library which want to solve irregular problems are involved in five major steps [120]:

- decompose the **distributed array** irregularly with the user provided information; The partitioner calculates how data arrays should be distributed.
- The newly calculated ***array distribution***s are used to decide how loop iterations are to be partitioned among processors. On distributing data, the runtime routines for this step determine on which processor each iteration will be executed.
- actual remapping of arrays from the old ***distribution*** to the new ***distribution***
- preprocessing needed for software ***caching***

CHAOS has implemented a set of optimization primitives in a library called ***PARTI***.

3 Berkeley's library

[152] states two basic techniques for implementing shared distributed data structures: replication and partitioning. Replication is supposed to give high throughput for read-only operations whereas partitioning has opposite characteristics. A library developed at Berkeley uses a combination of both (see ***Multipol***). The programming model is an event driven one, where each processor repeatedly executes a scheduling loop, which looks for work in one or more scheduling queues. [152] also represents a way to implement distributed data structures. A relaxed consistency model is applied and looks as follows: Each processor will see a consistent version of the data structure, but the operation does not always take effect on all processors simultaneously.

iteration loop transformation see ***irregular problems***

J

Jovian

abstract:

Jovian is an I/O library that performs optimizations for one form of **collective-I/O**. It makes use of a Single Program Multiple Data (***SPMD***) model of computation.

aims:

optimize performance of **distributed memory** parallel architectures that include multiple disks or disk arrays

implementation platform:

IBM SP1, ***Intel Paragon***

data access strategies:

regular and irregular access patterns (see ***irregular problems***)

portability:

The file operations are done through the native file system. Hence, the Jovian library is portable across multiple memory platforms, including networks of workstations [9].

related work:

PASSION, disk-directed I/O, CHAOS, PARTI

application:

Geographical Information Systems (GIS), Data Mining

people:

Robert Bennett, Kelvin Bryant, Alan Sussman, Raja Das, Joel Saltz

{robert, ksb, als, raja, saltz}@cs.umd.edu

institution:

Department of Computer Science, University of Maryland, College Park, MD 20742

http://www.cs.umd.edu/projects/hpsl/io/io.html

key words:

SPMD, **collective-I/O**, **out-of-core**, **distributed memory**, ***loosely synchronous***

details:

Jovian distinguishes between **global** and **distributed view**s of accessing data structures. In the **global view** the I/O library has access to the in-core and **out-of-core** data ***distributions***. What is more, **application process**es requesting I/O have to provide the library with a globally specified subset of the data structure. In contrast, in the **distributed view** the **application process** has to convert local in-core data indices into global **out-of-core** ones before making any I/O request [9]. The library consists of two types of processes: **application processes (A/P)** and **coalescing processes (C/P)** (similar to server processes in a DBMS). At link time there is no distinction between **A/P**s and **C/P**s [9]. The name **C/P** stems from the fact that coalescing I/O requests into a larger one can increase I/O performance. A user can determine which process will run the application and which will perform coalescing of I/O requests.

The implementation of collective read/write operations proceeds in several phases [9]:

- Request: Each **A/P** makes a Jovian I/O call to read or write. The library creates the required block requests from the various types of I/O calls and forwards these requests to a statically assigned **C/P**.
- Exchange: The requests that cannot be satisfied locally are forwarded to the **C/P** in a **collective communication** phase.
- Read and Return Blocks: The **C/P**s read the requested blocks from disk and send them directly to requesting **A/P**s.
- Send and Write Blocks: The **C/P**s create communication schedules to receive data from **A/P**s.

K

Kenal Square (KSR-2)

KSR-2 is a non-uniform access **shared memory machine**. See also ***Global Array (GA)***.

L

large request see ***RAID-I***

Last Recently Used see ***CCFS***

latency see ***parallel file system***

LFP (local file pointer) see ***SPIFFI***

LFS (Local File Server) see ***Cache Coherent File System (CCFS)***

LFS (Log-Structured File System) see ***RAID-II***

Light-weighted processes (LWP) see ***Cache Coherent File System (CCFS)***

Linda

Linda is a concurrent programming model with the primary concept of a **tuple space**, an abstraction via which cooperating processes communicate [55].

load utility see ***object-oriented database***

Local Array File (LAF) see ***PASSION***

local file pointer (LFP) see ***SPIFFI***

Local File Server (LFS) see ***Cache Coherent File System (CCFS)***

Local Placement Model see ***Global Placement Model (GPM)***

Local Server Invocation (LSI) see ***Hurricane operating system***

lock see ***Strict Two-Phase Locking***

Log-Structured File System (LFS) see ***RAID-II***

logical data locality see ***ViPIOS***

loosely synchronous

In a loosely synchronous model all the participating processes alternate between phases of computation and I/O [9]. In particular, even if a process does not need data, it still has to participate in the I/O operation (See also ***Two-Phase Method***.). What is more, the processes will synchronize their requests (**collective communication**).

LPM (Local Placement Model) see ***Global Placement Model (GPM)***

LSI (Local Server Invocation) see ***Hurricane operating system***

LWP (Light-weighted processes) see ***Cache Coherent File System (CCFS)***

M

many-to-many communication see ***CHANNEL***

mapped-file I/O

A contiguous memory region of an application's address space can be mapped to a contiguous file region on secondary storage. Accesses to the memory region behave as if they were accesses to the corresponding file region [94].

The advantages of mapped-file I/O [94]:

- little policy related information is embedded in access to file data
- secondary storage is accessed in the same fashion as other layers in the memory hierarchy
- low overhead
- exploiting the memory manager

The main disadvantage is that mapped file I/O can only be used for accessing disk files while ***I/O interfaces*** support also accesses to a file, terminal or network connection. The ***Hurricane File System*** has a library called ***Alloc Stream Facility (ASF)*** which refers to this problem.

Under paricular circumstances, mapped-file I/O can result in more overhead than read/write interfaces. E.g. writing large amounts of data past the end-of-file (EOF), modifying entire pages when data is not in the file cache [94].

MasPar MP-2 see ***CVL***

Massively Parallel Processor (MPP) see ***Grand Challenge Applications, Cache Coherent File System (CCFS), Pablo, PVM***

master client see ***Panda (Parallel AND Arrays)***

master node see ***Vesta***

master server see ***Panda (Parallel AND Arrays)***

master-slave see ***p4, PVM, SHORE***

mean-time-to-failure (MTTF) see ***RAID***

Memory-Style ECC (RAID Level 2) see ***RAID***

Mentat Programming Language see ***MPL***

message passing see ***Agent Tcl, CCFS, CHAOS, DDLY, distributed computing, Global Arrays, Intel iPSC/860 hypercub, Intel Paragon, Intel Touchstone, MIMD, MPI, MPI-2, p4, PASSION, PIOUS, PVM, Vesta, VIP-FS***

Message Passing Interface Forum (MPIF) see ***MPI, MPI-2***

Meta-Chaos see ***CHAOS***

meta-data see ***Galley, HiDIOS, HFS, Portable Parallel File System (PPFS), SPIFFI, Vesta***

metacomputing

Metacomputing defines an aggregation of networked computing resources, in particular networks of workstations, to form a single logical parallel machine [107]. It is supposed to offer a cost-effective alternative to parallel machines for many classes of parallel applications. Common metacomputing environments such as ***PVM***, ***p4*** or ***MPI*** provide interfaces with similar functions as those provided for parallel machines. These functions include mechanisms for interprocess communication, synchronization and ***concurrency control***, fault tolerance, and dynamic process management. Except of ***MPI-IO***, they do not support file I/O or serialize all I/O requests.

MIMD (Multiple Instruction Stream Multiple Data Stream)

MIMD is a more general design than ***SIMD***, and it is used for a broader range of applications [3]. Here each processor has its own program acting on its own data. It is possible to brake a program into subprograms which can be distributed to the processors for execution. Several problems can occur. For example, the scheduling of the processors and their synchronization. What is more, there will also be a need for more flexible communication than in a ***SIMD*** model. MIMD appears in two forms. First, with a private memory for each process - also referred to as **distributed memory** - and, second, with a **shared memory**. A **distributed memory** approach uses **message passing** for interprocess communication.

Mirrorde (RAID Level 1) see ***RAID***

mirroring see ***RAID***

MMP (Massively Parallel Processor) see ***Grand Challenge Applications***

moderate request see ***RAID-I***

MPI (Message Passing Interface)

abstract:

> In the last years, many vendors have implemented their own variants of the **message passing** paradigm, and it turned out that such systems can be efficiently and portably implemented. Message Passing Interface (MPI) is the de facto standard for **message passing**. MPI does not include one existing **message passing** system, but makes use of the most attractive features of them. The main advantage of the **message passing** standard is said to be 'portability and ease-of-use' [52]. MPI is intended for writing

message passing programs in C and **Fortran77**. The following passage is only based on the standard produced in May 1994 and does not inlcude new features from ***MPI-2***.

aims:

The goal of MPI was to develop a widely used standard for **message passing** programs, and to allow the communication of processes which are used by an MPI program.

data access strategies:

MPI is suitable for the use of ***MIMD*** (multiple instruction stream multiple data stream) programs as well as ***SPMD*** (single program multiple data). Blocking communication is the standard communication mode. A call does not return until the message data and envelope have been safely stored, and the send is free to access and overwrite the send buffer. MPI offers three additional communication modes that allow one to control the choice of the communication protocol. These three additional modes are: buffered mode, synchronous mode and ready mode.

In a blocking approach, a send command waits as long as a matching receive command is called by another process. Moreover, if the process returns from this procedure, the resources (such as buffers) can be reused (see ***data reuse***). On the other hand, in a nonblocking communication, a procedure may return before the operation completes, and before a user is allowed to reuse resources specified in the call.

portability:

I/O is a very important feature, but MPI (Version May 1994) does not support it (***MPI-2*** does !). ***MPI-IO*** [111] is based on the MPI standard and serves as a standard for parallel I/O within a **message passing** system. C, C++, Fortran

related work:

MPI-2, ***PVM*** (see ***PVM*** for a comparison of ***PVM*** and MPI)

people:

Jack Dongarra, David Walker (Conveners and Meeting Chairs); a detailed list of people is omitted here

dongarra@cs.utk.edu

institution:

MPI was developed by the **Message Passing Interface Forum (MPIF)** with participation from over 40 organizations.

`http://www.mpi-forum.org`

key words:

message passing, **distributed memory**, ***SIMD***, ***MIMD***

example:

```
#include "mpi.h"
main (argc, argv)
int argc,
    myrank,
    num_proc;
{
  MPI_Init (&argc, argv); // initialise MPI
  MPI_Comm_rank (MPI_COMM_WORLD, &myrank); // find rank
  MPI_Comm_size (MPI_COMM_WORLD, &num_proc); // find number of processes
  printf ('My process number is %d and %d processes are used.\n',
           myrank, num_proc);
  MPI_Finalize (); // terminate MPI computation
}
```

details:

1 The Basics of MPI

The easiest way of interprocess communication is the basic **point-to-point communication**, where two processes exchange their messages by the basic operations SEND and RECEIVE. Although MPI is a powerful and complex standard, it is possible to write programs which use just six basic functions [52].

`MPI_Init`	initiate an MPI computation
`MPI_Finalize`	terminate a computation
`MPI_Comm_size`	determine number of processes
`MPI_Comm_rank`	determine current process' identifier
`MPI_Send`	send a message
`MPI_Recv`	receive a message

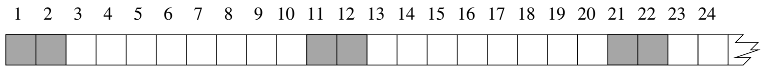


Figure 2.11: Noncontiguous Data Chunks in MPI

2 Derived Datatypes

In the former descriptions of **point-to-point communication**, MPI routines have only involved simple datatypes such as integers, reals, characters or arrays. Furthermore, contiguous buffers have to be used, i.e. one element of an array or the whole array has to be sent by a single `MPI_Send`. **Derived datatypes** allow to define noncontiguous elements to be grouped together in a message. This general mechanism allows one to transfer chunks of an array directly without copying noncontiguous elements into a buffer before being sent. A general datatype specifies two things:

- a sequence of basic datatypes
- a sequence of integer displacements

Such a pair of sequences is referred to as a **type map**. The displacement is not required to be distinct, positive, or in an increasing order. The basic datatypes in MPI are particular cases of a general datatype and can also be represented by a **type map**. For example, `MPI_INT` has the following **type map**: (int,0) with displacement 0 (no gap in between two adjacent integer values in the buffer or storage). All the other basic datatypes are similar.

For instance, an array consists of 50 elements and only the shaded elements (see Figure 2.11) are to be sent, i.e. the block size is 2 (two elements) and the stride is 10 (eight elements in between the shaded ones).

3 Collective Communication

In a **point-to-point communication** only two processes can compete in an information exchange process. One process sends data, and the other one receives that message. A **collective communication** is defined as a communication that involves a group of processes rather than only two. For instance, on process has to distribute an array to all other processes available. If a **point-to-point communication** is used, many send receive pairs have to be executed, and always the same information is transferred. For convenience, these operations can be done with one instruction by a broadcast command.

MPI offers the following functions for **collective communication** [52]:

- Barrier: Synchronises all processes.
- Broadcast: Sends data from one process to all processes.
- Gather: Gathers data from all processes to one process.
- Scatter: Scatters data from one process to all processes.
- Reduction operations: Sums, multiplies, etc., distributed data.

A collective operation means that all processes in a group have to execute the communication routine with the same parameters. This is an important difference between collective and **point-to-point communication**. In the latter, a receive has at least one different argument that is not the same in the send call: destination and source have to be adjusted. In a **collective communication** either a destination or a source has to be stated, depending on the function. Collective routines such as broadcast or gather have a single originating or receiving process called the root. In other words, instead of using a source and a destination, only a root process is defined. Another key argument is the **communicator** that defines the group of participating processes.

A **collective communication** provides stricter type matching conditions than a **point-to-point communication**. The amount of data sent must exactly match the mount of data specified by the receiver. What is more, a **collective communication** may or may not have the effect of synchronizing all calling processes (`MPI_Barrier` excluded), because collective routines return as soon as their participation in the **collective communication** is complete.

4 Groups and Communicators

Communicators in general provide an appropriate scope for all communication operations in MPI. They are divided into two kinds: **intra-** and **inter-communicator**s. A intra-**communicator** is used for operations within a group of processes. A group is defined as an ordered collection of processes, each with a rank. It is possible to create more than one group, each with its own intra-**communicator**. Processes can only communicate with their specified group. An **inter-communicator** allows a **point-to-point communication** between two independent groups.

MPI-2

abstract:

MPI-2 is the product of corrections and extensions to the original ***MPI*** Standard document. Although some corrections were already made in Version 1.1 of ***MPI***, MPI-2 includes many other additional features and substantial new types of functionality [104]. In particular, the computational model is extended by dynamic process creation and one-sided communication, and a new capability in form of parallel I/O is added (***MPI-IO***). (Note that every time when ***MPI*** is mentioned this dictionary refers to Version 1.0. Thus, if a passage refers to MPI-2, it explicitly uses the term MPI-2.)

aims:

see ***MPI*** and ***MPI-IO***

data access strategies:

see ***MPI*** and ***MPI-IO***

portability:

see ***MPI*** and ***MPI-IO***

related work:

MPI, ***MPI-IO***, ***PVM***, ***ROMIO***

people:

Many people have served on the **Message Passing Interface Forum**, but here only the primary coordinators are stated: Ewing Lusk, Steve Huss-Ledrerman, Bill Saphir, Marc Snir, Bill Gropp, Anthony Skjellum, Bill Nitzberg, Andrew Lumsdaine, Jeff Squyres, Arkady Kanevsky

institution:

Message Passing Interface Forum

http://www.mpi-forum.org

key words:

message passing, **distributed memory**, I/O interface, ***SIMD***, ***MIMD***

details:

In the following passages only the most significant new features of MPI-2 are discussed briefly, but the features from ***MPI*** Versions 1.0 and 1.1 are still valid and compatible.

1 Process Creation and Management

The static process model from ***MPI*** is extended, i.e. it is now possible to create or delete processes from an application after it has been started [104]. The work was influenced by ***PVM***, where such a process management is already included. However, resource control is not addressed in MPI-2, but it is assumed to be provided externally (e.g. by computer vendors, in case of tightly coupled systems, or by a third software package in case of a cluster of workstations). To sum up, MPI-2 does neither manage the parallel environment nor change the concept of **communicator**s.

2 One-Sided Communications

The ***MPI*** communication mechanisms are extended by **Remote Memory Access (RMA)** which allows that one process can specify all communication parameters for both the sending and the receiving side. In other words, communication routines can be completed by a single process. Moreover, this mechanism avoids the need for global computations or explicit polling [104]. In more detail, the following three communication calls are provided: `MPI_PUT` (remote write), `MPI_GET` (remote read) and `MPI_ACCUMULATE` (remote update). It is supposed that these functions should enable fast communication mechanisms provided by various platforms.

3 Extended Collective Operations

Shortly, there a some extensions regarding the collective routines in ***MPI***, additional routines for creating intercommunicators and two new collective routines: a generalized all-to-all and exclusive scan. What is more, "in place" buffers can be specified.

4 External Interfaces

MPI-2 allows developers to layer on top of MPI-2 by means of generalized requests. What is more, users should be able to define nonblocking operations which occur asynchronously and, hence, concurrently with the execution of the user code.

5 I/O Support

The previous versions of ***MPI*** did not include any support for parallel I/O. MPI-2 picks up the basic ideas from ***MPI-IO*** and includes them into the MPI-2 standard.

Finally, MPI-2 also includes C++ bindings and discusses **Fortran 90** issues.

MPI-IO

abstract:

Despite the development of ***MPI*** as a form of interprocess communication, the ***I/O problem*** has not been solved there. (Note: ***MPI-2*** already includes I/O features.) The main idea is that I/O can also be modeled as **message passing** [111]: writing to a file is like sending a message while reading from a file corresponds to receiving a message. Furthermore, MPI-IO supports a high-level interface in order to support the partitioning of files among multiple processes, transfers of global data structures between process memories and files, and optimizations of physical file layout on storage devices [111].

aims:

The goal of MPI-IO is to provide a standard for describing parallel I/O operations within an ***MPI*** **message passing** application. Other goals [111]:

- targeted primarily for scientific applications
- favors common usage patterns over obscure ones
- the features are intended to correspond to real world requirements

- allows the programmer to specify high level information about I/O
- performance rather than just functionality

data access strategies:

MPI-IO provides three different access functions including positioning, synchronizm and coordination. Positioning is accomplished by explicit offsets, individual file pointers, and ***shared file pointer***s. As for synchronizm, MPI-IO provides both synchronous and asynchronous (blocking, nonblocking respectively) versions. Moreover, MPI-IO supports collective as well as independent operations.

portability:

MPI-IO provides two different header files and, hence, can be used in the C and in the Fortran programming languages. ***MPI*** **derived data types** are used.

C, C++, Fortran

related work:

MPI-2, ***ROMIO***

people:

a list of people is omitted here

`mpi-io-requests@nas.nasa.gov`

institution:

The MPI-IO proposal was made by the **MPI-IO committee** which consists of people from the following institutions:

IBM T.J. Watson Research Center

NASA Ames Research Center

Lawrence Livermore National Laboratory (LLNL)

Argonne National Laboratory (ANL)

`http://lovelace.nas.nasa.gov/MPI-IO`

key words:

MPI, high-level interface, **message passing**, I/O features, ***MPI-2***

example:

In the example, processes access data from a common file.

```
#include "mpi.h"
#include "mpio.h"
#define SIZE 1024

main (argc, argv)
int argc;
char *argv[];
{
  int rank, buf[SIZE];
  MPIO_File fh; // file handle
  MPIO_Offset offset; // offset in file
  MPIO_Status status;

  MPI_Init (&argc, &argv);
  MPIO_Init (&ragc, &argv);  // initialize MPI-IO
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);

  // each process opens a common file called 'test'
  MPIO_Open (MPI_COMM_WORLD, "test", MPIO_CREATE | MPIO_RDWR,
             MPIO_OFFSET_ZERO, MPI_INT, MPI_INT, MPIO_HINT_NULL, &fh);

  // perform computation

  // write buffer
  offset = rank * SIZE;
  MPIO_Write (fh, offset, buf, MPI_INT, SIZE, &status);

  // perform computation

  // read the previously stored data in to buffer
  MPIO_Read (fh, offset, buf, MPI_INT, SIZE, &status);

  // perform computation

  MPIO_Close (fh);
  MPIO_Finalize ();
  MPI_Finalize ();
}
```

details:

MPI-IO has six basic functions that are sufficient to write many useful programs [111].

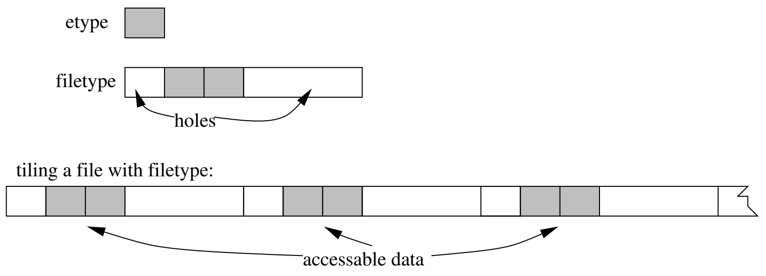


Figure 2.12: Tiling a file using a filetype

`MPIO_Init`	initialize MPI-IO
`MPIO_Open`	open a file
`MPIO_Read`	read data from a particular location in the file
`MPIO_Write`	write data to a particular location in the file
`MPIO_Close`	close a file
`MPIO_Finalize`	terminate MPI-IO

MPI-IO should be as ***MPI*** friendly as possible [111]. Like in ***MPI***, a file access can be independent (no coordination between processes takes place) or collective (each process of a group associated with the **communicator** must participate in the collective access). What is more, ***MPI*** **derived data types** are used for the data layout in files and for accessing shared files. The usage of **derived data types** can leave holes in the file, and a process can only access data that falls under holes (see Figure 2.12). Thus, files can be distributed among parallel processes in disjoint chunks.

Since MPI-IO is intended as an interface that maps between data stored in memory and a file [111], it basically specifies how the data should be laid out in a virtual file structure rather than how the file structure is stored on disk. Another feature is that MPI-IO is supposed to be interrupt and thread safe.

MPI-IO committee see ***MPI-IO***

MPL (Mentat Programming Language)

Mentat is an object oriented parallel processing system [79]. MPL is a programming language based on C and used to program the machines MP-1 and MP-2 [38].

Multi-user Multimedia-on-Demand Server

At the heart of this system is a high-performance server which is a **massively parallel processor (MPP)** optimized for fast parallel I/O. The sever is connected to high-speed wide-area-network with ATM switches. The remote clients are computers with tens of megabytes of main memory and hundreds of megabytes of secondary storage [66].

multifiles see ***design of parallel I/O software, ParFiSys (Parallel File System)***

Multipol

abstract:

Multipol is a publicly available [153] library of distributed data structures designed for irregular applications (see ***irregular problems***). Furthermore, it contains a thread system which allows overlapping communication latency with computation [154].

aims:

development of irregular parallel applications

implementation platform:

Thinking Machines **CM-5**, ***Intel Paragon***, **IBM SP1**; in the future [154]: network of workstations

data access strategies:

A communication layer provides portable communication primitives for bulk-synchronous (put and get operations) and asynchronous communication (to start a thread on a remote processor).

related work:

PARTI provides compiler analysis and runtime support, but it is not effective in asynchronous applications due to the dynamic change of computation patterns [154].

TAM (Threaded Abstract Machine)

application:

symbolic applications that require dynamic irregular data structures written for distributed and some for **shared memory** multiprocessors [153]:

- Knuth-Bendix (used in automatic theorem providing systems based on equations called "rewrite rules")
- Term Matching, Eigenvalue
- Groebner Basis (completion procedure to manipulate polynomials)
- Phylogeny Problem (problem of determining the evolutionary history for a set of species - fundamental in molecular biology)
- Tripuzzle (problem to compute the set of all solutions to a single player board game)

people:

Chih-Po Wen, Soumen Chakrabarti, Etienne Deprit, Arvind Krishnamurthy, Katherine Yelick, Jeff Jones

`{cpwen, soumen, deprit, yelick, arvind, jjones}@cs.berkeley.edu`

institution:

University of California, Berkeley, CAL 94720, USA

`http://HTTP.cs.Berkeley.EDU/Research/Projects/parallel/castle/multipol`

key words:

I/O library, **distributed memory**, distributed data structures, ***irregular problems***

details:

The logically shared data structure is physically distributed among the processors. Since **distributed memory** architectures do not support a shared address space, computation migration (see also ***Agent Tcl***) can be used to save communication costs [154], i.e. the operation migrates to the processor where the bucket resides; this avoids ***data prefetching***, however, the latency may be longer. A communication layer called 'active messages' can be applied for computation migration. A **split-phase interface** is provided for operations that may require communication. Such an operation returns after having done local computations, but never waits for communications to complete. Therefore, the caller has to check for completion.

MTTF (mean-time-to-failure) see ***RAID***

N

Name Server see ***Hurricane File System (HFS)***

NAP see ***Network-Attached Peripherals***

nCUBE

The proposed file system for the nCUBE is based on a two-step mapping of a file into the **compute node** memories [113], where the first step provides a mapping from subfiles stored on multiple disks to an abstract data set, and the second step is mapping the abstract data set into the **compute node** memories. One drawback is that it does not provide an easy way for two **compute node**s to access overlapping regions of a file [113]. The throughput is examined in [84]. See also ***PASSION***.

nested patterns see ***disk-directed I/O***

Network-Attached Peripherals (NAP)

NAP make storage resources directly available to computer systems on a network without requiring a high-powered processing capability [134]. This makes it possible for a single network-attached control system such as **HPSS (High-Performance Storage System)** to manage access to the storage devices without being required to handle the transferred data. In particular, **HPSS** is capable of coordinating concurrent I/O operations over a non-blocking network fabric to achieve very high aggregate I/O throughput. See also ***Scalable I/O Facility (SIOF)***.

NAP features [134]:

- I/O operations performed under the control of a master via an authenticated control path
- data transfers directly with the client to obtain optimal performance
- access via conventional network media using conventional protocols to provide a reliable data path

Non-Redundant (RAID Level 0) see ***RAID***

O

object-oriented database (OODB)

An OODB requires a **load utility** in order to be able to load much data quickly from disk into main memory, especially if neither the data nor the id map fits in memory. [151] states two large databases in the scientific community:

- The Human Genome Database
- The climate modeling project

[151] presents some algorithms for a good **load utility**, one is the partitioned-list where random data access can be eliminated by writing the id map to disk as a persistent list, and using a hash join to perform lookups. In particular, **virtual memory** structures or persistent B+ trees are applied.

A database system based on workstations is also implemented using a **client-server** architecture [53]. As for the **client-server *EXODUS***, it utilizes a **page server architecture**. Here clients interact with servers by sending requests for specific database pages or groups of pages [22].

object-oriented data base management system (OODBMS) see ***SHORE***

OCAD (Out-of-core Array Descriptor) see ***PASSION***

OCLA (Out-of-core Local Array) see ***PASSION***

OFS (Open File Server) see ***Hurricane File System (HFS)***

one-to-one communication see ***CCFS***

Open File Server (OFS) see ***Hurricane File System (HFS)***

operating system components

[10] depicts operating system components that offer coverage over three critical requirements for message-based massively parallel multicomputers:

- networking
  - **Intranetworking** enables communication between processors that are part of the same multicomputer and may be within or between parallel programs. There are two distinctive solutions: unprotected and protected communication.
  - **Internetworking** enables communication between remote processors connected by networks such as Ethernet, HIPPI, FDDI and ATM.
- memory management includes three techniques: ***Shared Virtual Memory***, ***Remote Memory Servers*** and ***Checkpointing***
- file and object store: It will be comprised by three major tasks: Firstly, defining and implementing a standard application interface to both the Intel ***Parallel File System*** (***PFS***) and the IBM ***Vesta*** Parallel File System. Secondly, it will be explored whether a parallel, persistent object store can provide satisfactory performance to ***HFS*** applications. Finally, design and evaluation of alternative strategies for ***prefetching*** files and objects from tertiary storage into secondary storage.

OOC (out-of-core) see ***API, irregular problems, PASSION, Two-Phase Method (TPM), ViPIOS***

OODB see ***object-oriented data base***

OODBMS (object-oriented data base management system) see ***SHORE***

Opal see ***CHAOS***

OPT++

abstract:

OPT++ is an object-oriented tool for Extensible Database Query Optimization to simplify the implementation, extension and modification of an optimizer [76]. Moreover, OPT++ uses object-oriented programming tools from C++, even the search strategy is a class. The optimizer consists of three components: a Search Strategy, a Search Space and the Algebra.

aims:

It should be easy to add new operators and execution algorithms, experiments should be enabled and flexibility should not influence the efficiency in a negative way.

implementation platform:

experiments were executed on a Sun **SPARC**-10/40

related work:

EXODUS Optimizer Generator, Volcano Optimizer Generator

people:

Navin Kabra, David J. DeWitt

{navin, dewitt}@cs.wisc.edu

institution:

Computer Science Department, University of Wisconsin, Madison, WI 53706, USA

http://www.cs.wisc.edu/shore/

key words:

object-oriented database, query optimizer

example:

see Figure 2.13

details:

The query is represented by an operator [76] tree where each node denotes a logical query

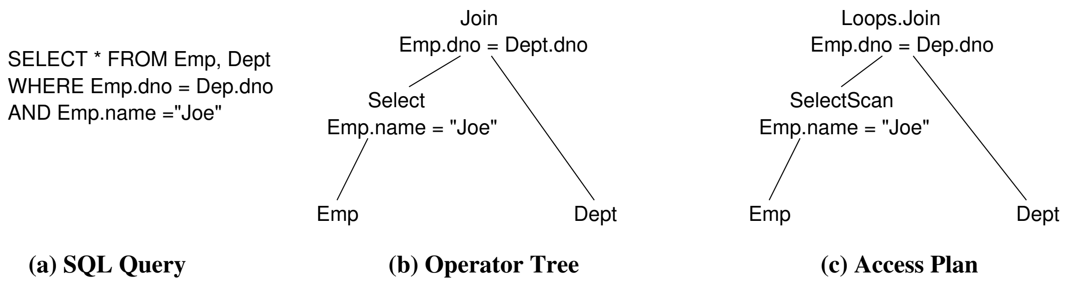


Figure 2.13: Query Representation in OPT++

algebra operator as an input. The operators in the tree are replaced by the algorithms and yield an access plan or execution plan. OPT++ has implemented a number of search strategies including Bottom-up Search, Transformative Search and Random Search Strategies.

Ordered Access see ***Portable Parallel File System (PPFS)***

OSF/1

OSF/1 is the operating system for the ***Intel Paragon*** multicomputer. See also ***PFS***.

out-of-core (OOC) see ***C*, disk-directed I/O, irregular problems, Jovian, Panda, PIM, PASSION, supercomputing applications, TPM, ViC*, Vienna Fortran***
Out-of-core Array Descriptor (OCAD) see ***PASSION***
Out-of-core Communication Method see ***PASSION***
Out-of-core Local Array (OCLA) see ***PASSION***
Overlapping or Disjoint Access see ***Portable Parallel File System (PPFS)***

P

p4

p4 is a library of macros and subroutines developed at ***ANL*** for programming parallel machines [55]. It supports **shared memory** and **distributed memory**, where the former is based on monitors and the later is based on **message passing**. Like in ***PVM***, p4 offers a **master-slave** programming model.

P+Q Redundancy (RAID Level 6) see ***RAID***

Pablo

abstract:

Pablo is a massively parallel, **distributed memory** performance analysis environment to provide performance data capture, analysis, and presentation across a wide variety of scalable parallel systems [124]. Pablo can help to identify and remove performance **bottleneck**s at the application or system software level. The Pablo environment includes software performance instrumentation, graphical performance data reduction and analysis, and support for mapping performance data to both graphics and sound [124]. In other words, Pablo is a toolkit for constructing performance analysis environments.

aims:

- portability
- **scalability**: The performance of a system can be increased by adding processors.
- extensibility: The user should be able to interact with the data to change the types of data analysis and to add new analysis as needed [124]. What is more, the system should also be sufficient for different kinds of users including novice, intermediate and expert.

implementation platform:

Intel iPSC/860 hypercube and Intel's NX/2 operating system

related work:

Prism and NV for CM-Fortran, Forge90 for F90 and ***HPF***, MPP-Apprentice performance tool for C, Fortran, **Fortran90**

people:

Daniel A. Reed, Ruth A. Aydt, Tara M. Madhyastha, Roger J. Noe, Keith A. Shields, Bradley W. Schwartz, Jhy-Chun Wang

`{reed, aydt}@cs.uiuc.edu`

institution:

Department of Computer Science, University of Illinois, Urbana, IL 61801

`http://bugle.cs.uiuc.edu/Projects/IO/io.html`

key words:

performance analysis

example:

The sample file from [5] contains performance trace data such as a single Stream Attribute, two Record Descriptors, and four Record Data instances (details below). The Stream Attribute gives information about the trace file (created on 1 Febr. 1992), and the Record Descriptors define the event types "message send" and "context switch" (labeled with #1 and #2).

```
SDDFA
/*
 *"run date" "February 1, 1992"
 */ ;;

#1: // "event" "message sent to one ore more processes"
"message send" {
  double "timestamp";
  int "sourcePE"; // "Source" "Processor Sending Message"
  int "destinationPE"[]; // "Destination" "Processor(s) Receiving Message"
  int "message length" // "Size" "Message length in bytes"
};;

#2;
"context switch" {
  double  "timestamp";
  int  "processor";
  char    "processName[]";
};;

"context switch" {100.150000, 2, [8] { "file i/o" } };;

"message send"   {101.100000, 0, [4] {1, 3, 5, 7, }, 512};;

"message send"   {102.150000, 7, [1] {1}, 1012};;

"context swtich" {108.000000, 4, [4] {"idle"} };;
```

details:

The Pablo instrumentation software has three components [124]:

- graphical interface for interactively specifying source code instrumentation points
- C and Fortran parsers that emit source code with embedded calls to a trace capture library
- trace capture library that records the performance data

Since there is no standard data format for processing trace data from sequential or parallel computer systems [124], the **Pablo Self-Describing Data Format (SDDF)** has been developed. It is a trace description language for specifying both the structure of data records and data record instances. The format can be represented in ASCII and binary format. Each **SDDF** file contains a flag indicating the byte ordering used by the file [5]. There exists also a library of C++ classes that provides an interface to the data stored in **SDDF** files.

A correct **SDDF** file can only be created if the **SDDF** syntax rules are obeyed. The **SDDF** meta-format, which is supposed to be the key source of flexibility [1], has four classes of records [5]:

- Command: conveys the action to be taken e.g. `%17;`
- Stream Attribute: gives information pertinent to the entire file e.g.:

```
/*
 *"Stream Attribute: This is just information"
 */;;
```

- Record Descriptor: declares the record structure (see program fragment followed by #1)
- Record Data: encapsulates data values
  E.g.: `{"context switch" {100.150000, 2, [8] {"file i/o"} };;`

Each file begins with a header identifying the file format: SDDFA represents an ASCII file and SDDFB a binary file. A record in the file is referred to as a packet. The four types of these packets are stated above. A packet consists of a header and a body (except command packets). [5] gives details on the usage of the **SDDF** Interface Library. A description of the classes and methods is given in [64].

Pablo Self-Describing Data Format (SDDF) see ***Pablo***
page server architecture see ***object-oriented database (OODB)***
paged translation table see ***PARTI***

Panda (Persistence AND Arrays)

abstract:

Panda is a library for input and output of multidimensional arrays on parallel and sequential platforms [131, 132]. Panda provides easy-to-use and portable array-oriented interfaces to scientific applications, and adopts a server-directed I/O strategy to achieve high performance for collective I/O operations.

aims:

Provide easy-to-use, portable, and high performance I/O support for synchronized collective I/O operations in SPMD-style application programs on workstation clusters and distributed memory multiprocessor platforms [30].

implementation platform:

Panda currently has been ported on a wide range of platforms:
Intel iPSC/860 (Intel CFS),
IBM SP2 (AIX JFS file system and IBM PIOFS parallel file system),
Intel Paragon (Intel ***PFS***), SGI Origin 2000 (SGI xFS),
Cray T3E (Unix File System),
SUN workstations (SunOS),
HP workstations with an FDDI interconnect (HP-UX),
Windows NT PC clusters with a Myrinet interconnect (WinNT file system)

data access strategies:

applications are closely synchronized during I/O [128] support for blocking **collective-I/O** [30]

Panda overlaps its file system I/O activities with its internal communication and computational activities.

Panda uses ***MPI*** as its internal communication mechanisms to move data between clients and servers.

In order to achieve ease-of-use and application portability Panda uses an array-oriented high level interface. Furtermore, high performance is gained by the usage of the **server-directed I/O** strategy to form long, sequential requests whenever possible.

portability:

Panda uses ***MPI*** for interprocess communication

related work:

two-phase I/O, ***disk-directed I/O***, ***Two-Phase Method*** (***PASSION***)

Hierarchical Data Format (HDF)

POSTGRES DBMS is enhanced: focus on read-only application that can take advantage of the query capabilities of a DBMS [129]

ViPIOS, ***HPF*** (hints to the compiler with `ALIGN` and `DISTRIBUTE`), ***Vesta***

application:

computational fluid dynamics (CFD) at NAS (airflow over aircraft and spacecraft)

the flow solver is written in an ***SPMD*** style in Fortran using explicit **message passing**

people:

Current group members:

Marianne Winslett (group leader), Y. Chen, Y. Cho, S. Kuo, J. Lee, and K. Motukuri.

`{winslett, ying, ycho, s-kuo, jlee17, motukuri}@bunny.cs.uiuc.edu`

Past group members:

K.E. Seamons, M. Subramaniam, P. Jones, and J. Jozwiak.

institution:

Computer Science Department, University of Illinois, Urbana, Illinois 61801, USA

`http://drl.cs.uiuc.edu/panda/`

key words:

server-directed I/O, ***SPMD***, high-level interface, ***MPI***, database, chunking, **distributed memory**

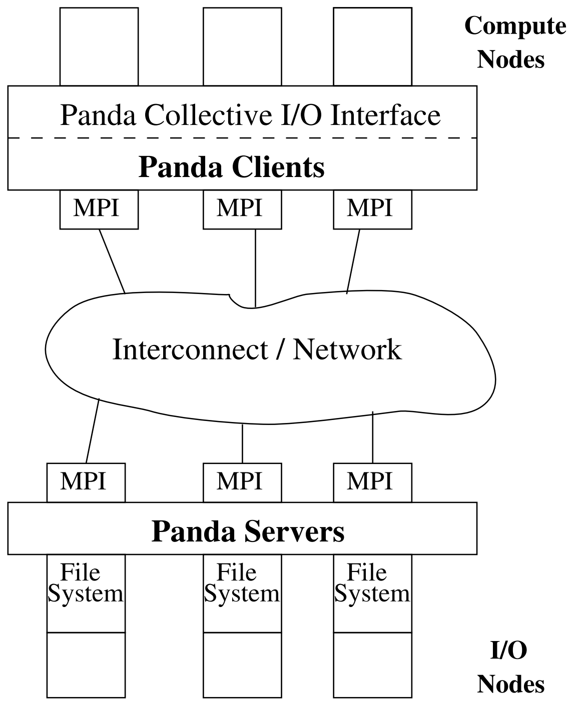


Figure 2.14: Panda Server-Directed I/O Architecture

details:

Panda combines three techniques in order to obtain performance [128]:

- storage of arrays by subarray chunks in main memory and on disk
- high-level interfaces to I/O subsystems
- use of ***disk-directed I/O*** to make efficient use of disk bandwidth

Array chunking can improve the locality of computation on a processor, and improve I/O performance. High-level interfaces are considered to be flexible, easier to be used by programmers and give applications better portability [128].

1 Internal Architecture

Nodes in Panda can be classified into **compute node**s (Panda clients) and **I/O node**s (Panda servers, see Figure 2.14). Panda uses the **server directed-I/O** (i.e. **I/O node**s direct the flow of I/O requests) strategy to handle application I/O requests [128]. Panda allows to distribute arrays across **compute node**s using ***HPF*** data ***distribution*** semantics.

The I/O request handling is carried out as follows: When an I/O request is issued, one **compute node** informs the servers of the upcoming I/O request, including the array distribution information, array sizes, and array ranks [98, 138]. The servers digest the information and make plans to handle the upcoming requests. For write requests, once each server determines what data is his responsibility, it gathers data in a large I/O buffer and issues the file system request to write out the entire buffer. The reverse is used for reads. Communication between servers does not take place during plan formation or while array data is gathered or scattered to the clients. The same is true for clients. At the end of I/O request handling, all the servers synchronize, and one of the servers, the master server, informs the master client who in turn, informs all other clients about the completion of the I/O requests.

On platforms with fast interconnect and slow disks, e.g., **IBM SP**s, the Panda performance is limited by the peak throughput of the underlying file system. However, on platforms with relatively fast disks, but slow interconnects, such as an FDDI connected HP workstation cluster, the performance can be bottlenecked by the underlying communication system. Panda currently uses different optimization algorithms to optimize I/O performance for different platforms.

2 Scalable Message Passing

Each client computes which servers it should communicate with in order to avoid an additional message to be exchanged [30]. Thus, the clients require certain information concerning the entire I/O request.

3 Panda and the Database Approach

Recent commercial database management systems (DBMS) are not applicable to scientific application due to the lack of many characteristics and facilities required in a computation intensive environment [130]. The members of Panda explore the use of flexible chunked array storage formats to store arrays on disk [130].

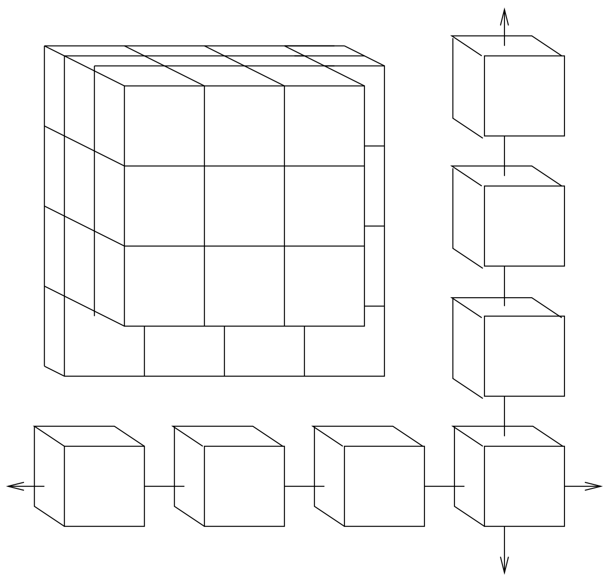


Figure 2.15: 3-D array divided into multidimensional chunks

An array has to be divided into chunks and can be stored contiguously on disk (see Figure 2.15). A chunk number is used to compute the correct chunk offset. What is more, chunks may be of different sizes within a single array, and can be stored in a packed or unpacked fashion.

A chunk can as well be divided into subchunks or can form a multidimensional array of lower rank [131], which can be referred to as array of chunks. Ideally, each processor will work on a single chunk at a time. The low-level interface allows reads and writes of logical chunks of an array. In contrast, the high-level interface supports the declaration of certain arrays to be part of checkpoints, time step outputs or restarts. The information about the chunk locations is stored in Panda's schema files which can be used for the subsequence data accesses.

Paradise (PARAallel Data Information SystEm)

The goal is to apply an object-oriented and parallel database approach of ***EXODUS***. Paradise supports storing, browsing, and querying of geographic data sets [47]. Moreover, an extended-relational database model is used. ***SHORE*** serves as the storage manager for persistent objects. The ***Intel Paragon*** was used for some experiments.

parallel file system

A parallel file system tries to eliminate the I/O **bottleneck** by logically aggregating multiple independent storage devices into a high-performance storage subsystem [108]. The bandwidth can be increased by **independent disk addressing** (the file system can access data conconcurrently from different files) and **data declustering** (a single file can be accessed in parallel).

Figure 2.16 depicts a very common model for a general parallel file system. Here the file system is implemented over a set of independent **I/O node**s that can be accessed by a set of **compute node**s via a high-speed interconnect. On each of the **I/O node**s resides an **I/O daemon** which implements general data access methods, disk scheduling, ***caching***, ***prefetching*** etc. In contrast, library routines linked with applications reside on each **compute node**. These libraries implement the file system interface and manage all communication with the **I/O node**s.

The following passage describes the minimal event sequence required for data access [108]: Firstly, it is determined which **I/O node**s contain the data to be accessed. In order to determine this, messages are sent to the **I/O daemon**s requesting that data to be read or written. Each daemon satisfies each request independently and sends a reply. The library function returns control to the caller only if all results have been collected. ***Concurrency control*** mechanisms have to be employed in order to guarantee correct transactions. However, ***concurrency control*** negates some of the performance benefits of **data declustering** [108].

A parallel file system requires special research issues: In detail, traditional measurements of I/O performance are **throughput** and **latency**. **Latency** is the time elapsed between issuing a request and completing it whereas **throughput** is the average number of requests completed over a period of time. It can be stated that general improvements in I/O performance do not necessarily imply improvements in both **throughput** and **latency**. The two

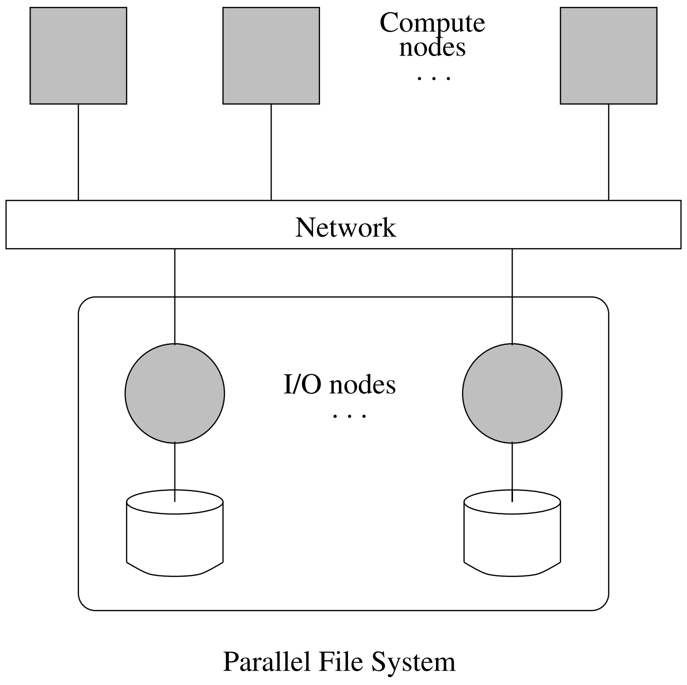


Figure 2.16: Generic Parallel File System Architecture

basic methods to minimize the effects of **latency** are latency avoidance and latency tolerance.

Another way of avoiding **latency** of physical I/O operations is to cache frequently used data. Furthermore, the location of the cache is an important issue. Whereas in a conventional memory hierarchy a cache is located on the bus between a fast and a slow memory, in a parallel I/O system the boundary is distributed to access parallel secondary storage and parallel processors. In addition, one can distinguish between a global, client and server cache. Striping, ***clustering*** and ***distribution*** are important terms regarding data ***distribution***. In order to increase the mean time to failure ***RAID***s are used. [7] presents some data on analyzing distributed file systems.

Parallel File System see ***PFS***

parallel I/O interface see ***I/O interfaces***

ParClient see ***ParFiSys (Parallel File System), Cache Coherent File System (CCFS)***

ParFiSys (Parallel File System)

abstract:

ParFiSys was developed to provide I/O services for a ***General Purpose MIMD machine (GPMIMD)***. It was named ***CCFS*** in earlier projects. ParFiSys tries to realize the concept of "minimizing porting effort" in the following way [27]:

- standard **POSIX** interface
- parallel services are provided transparently, and the physical data ***distribution*** across the system is hidden
- a single name space allows all the user applications to share the same directory tree

aims:

provide I/O services to scientific applications requiring high I/O bandwidth to minimize application porting effort, and to exploit the parallelism of a generic **message passing** multicomputer

implementation platform:

GPMIMD

developed on a UNIX multiprocessor environment; operational on the following environments: **Transputer T800** and **T9000** multiprocessors, ***IBM/SP2*** and UNIX multiprocessors.

data access strategies:

- Segmented files: Each file is composed of one or more segments and uses a file pointer with the items "segment number" and "segment internal address".
- Multifiles: A **multifile** can be accessed in parallel as a set of subfiles.
- Global files: provides file pointer sharing
- **Space preallocation**: Resources for a file can be allocated in advance.

portability:

MPI-IO can be integrated in ParFiSys related work:

MSS (Mass Store System), DPU (Data Parallel UNIX), ***CFS***, **CM-5** sfs, ***Vesta***, ***Galley***

people:

J. Carretero, F. Perez, P. de Miguel, F. Garcia, L. Alonso, F. Rosales

{jcarrete, fperez, pmiguel, fgarcia, lalonso, frosal}@fi.upm.es

institution:

Facultata de informatica, UPM (Universidad Politechnica de Madrid), Campus Monteganceda, E-28660 Boadilla des Monte, Madrid, Spain

http://laurel.datsi.fi.upm.es/ gp/parfisys.html

key words:

file system, ***GPMIMD***, data ***distribution***, **MPP**, **message passing**

details:

Since ParFiSys is the successor of ***CCFS*** and there are only a few differences (mainly concering cache coherence policies), the correspondence is stated here:

- **ParClient** corresponds to **CLFS**
- **ParServer** corresponds to **LFS**
- **ParDisk** corresponds to **CDS**

1 Architecture and Design

The architecture is divided into two levels called **file services** and **block services**. In the first level there are **ParClient**s (one on each **PN, processing node**) which translate user requests into logical block requests, and the **ParServer**s (on each **ION**, **I/O node**) in the second level interact directly with the I/O devices located on their own **ION** (see Figure 2.17).

ParFiSys uses following three techniques to translate user's I/O requests into orders of the physical system: data ***distribution***, transparent parallel access and operation grouping. In addition, the performance is increased by internal features such as ***read ahead***, **flush ahead**,

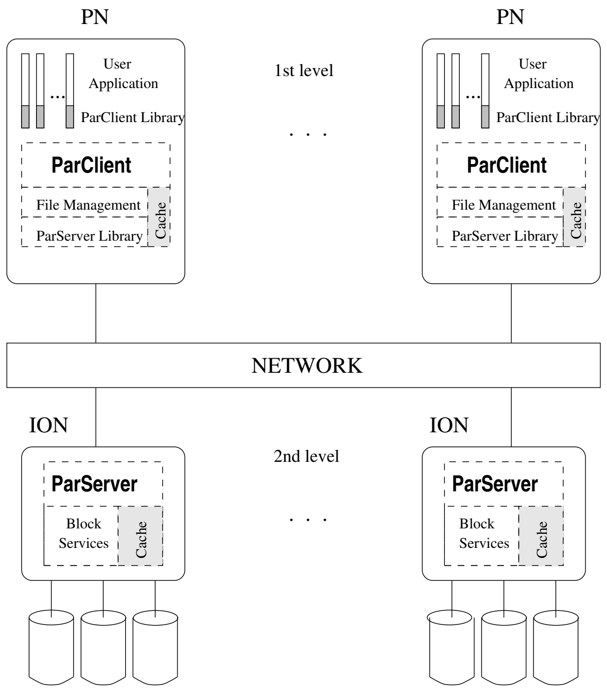


Figure 2.17: ParFiSys Architecture

extensive data ***caching*** and resource preallocation. In order to obtain data ***distribution***, ParFiSys requires four steps to translate user logical byte addresses into internal physical block addresses:

- mapping user logical bytes to ParFiSys logical blocks
- mapping ParFiSys logical blocks to physical **I/O node**s
- mapping ParFiSys logical blocks of each **I/O node** to physical devices
- mapping ParFiSys logical blocks to device physical blocks

The first level in ParFiSys (the **ParClient**), provides **file services** that can be obtained in two ways by either linked libraries or by a **message passing** library. A linked library is preferred for a parallel machine, where a single user is usually expected per **processing node**. A **message passing** library is preferred in distributed systems, where requests can be sent to the **ParClient** by several users. In a linked library architecture the **ParClient** has to be present on every **processing node (PN)** requesting I/O whereas the **message passing** approach allows remote users for a **ParClient**. The task of the **ParClient** is to translate user addresses into logical blocks establishing the connections with the **ParServer**s, handle all communication through a high performance I/O library called **ParServer** Library. **I/O server**s are contacted with via **message passing**. The **ParServer**s, located at the **I/O node**s, deal with logical block requests and translate them to the logical secondary storage devices.

2 Data Distribution

A very generic distributed partition allows to create several types of file systems on any kind of parallel I/O system. Such a distributed partition has a unique identifier, physical layout, etc. The physical layout is represented by the following tuple and describes the set of **I/O node**s, controllers per node and the devices per node [26]:

$$(\{NODE_n\}, \{CTLR_c\}_n, \{DEV_d\}_{nc})$$

ParFiSys supports three kinds of predefined file systems on the partition structure [28]:

- UNIX-like non-distributed file systems, where 'NODE'=1, 'CTLR'=1 and 'DEV'=1
- Extended distributed file systems with sequential layout, where 'NODE'=a, 'CTLR'=b and 'DEV'=c
- Distributed file systems striped with cyclic layout, where 'NODE'=a, 'CTLR'=b and 'DEV'=c. Here blocks are distributed through the partition devices in a **round-robin** fashion.

3 Data Access

Since most file systems have drawbacks because of using the logical block as the basic unit [28], following cache management algorithms are included:

- grouped management of the mapping and block I/O steps of each request
- grouping of several independent I/O requests

The mapping is established in a UNIX-like fashion and uses direct blocks, single indirect blocks, double indirect blocks and triple indirect blocks. In order to optimize the mapping a fixed number of blocks is premapped in advance to the next user request.

On mapping the user buffer to file system blocks, the whole block list is searched in the **ParClient** cache with a single seek operation. A present block (found in the cache) is immediately copied to the user space whereas absent blocks have to be requested through the **ParServer** library. The requesting is done concurrently to each **ParServer**, overlapping I/O and computation.

4 Resource Management
Allocating a big number of blocks can be very time consuming. Therefore, preallocation algorithms obtain a predetermined number of free resources whenever possible and yield in a major increase of the performance. However, a new problem named "resource liberation" is introduced. Resource preallocation has to be balanced in order to allow unused resources also to be used by other **ParClient**s.

parity logging see ***RAID***
parity stripe see ***RAID-I***
ParServer see ***ParFiSys (Parallel File System)***
ParSet (Parallel Set) see ***SHORE***
ParSet Sever (PSS) see ***SHORE***

PARTI (Parallel Automated Runtime Toolkit at ICASE)

abstract:

PARTI is a subset of the ***CHAOS*** library [120] and specially considers ***irregular problems*** that can be divided into a sequence of concurrent computational phases. The primitives enable the ***distribution*** and retrieval of globally indexed, but irregularly distributed data sets over the numerous local processor memories. What is more, it should efficiently execute unstructured and block structured problems on **distributed**

memory parallel machines [139]. The PARTI primitives can be used by parallizing compilers to generate parallel code from programs written in ***data parallel*** languages [43].

aims:

Ease the implementation of computational problems on parallel architectures by relieving the user low-level machine specific issues [139].

implementation platform:

Much work was done for the ***Intel Touchstone Delta***, but also for the ***Intel iPSC/860 hypercube***.

data access strategies:

PARTI uses the means of communications offered at Intel, namely **Intel forced message types**. Non-blocking receive calls (Intel irecv), which are posted before data is sent, are used. Synchronization messages check the consistency of posted messages.

related work:

Fortran D, ***HPF***, ***Vienna Fortran***

application:

An application was chosen from the domain of computational fluid dynamics (CFD). A serial code developed at the NASA Research Center was devoted to solve the thin-layer Navier-Stokes equation.

people:

Alan Sussman, Joel Saltz, Raja Das, Mustafa Uysal, Yuan-Shin Hwang, S Gupta, Dimitri Mavriplis, Ravi Ponnusamy

`{als, saltz, raja, uysal, shin}@cs.umd.edu`

institution:

Department of Computer Science, University of Maryland, College Park, MD 20742, USA

`http://www.cs.umd.edu/projects/hpsl.html`

key words:

irregular problems, **distributed memory**

details:

Each element in a **distributed array** is assigned to a particular processor. If another processor wants to access these data, it has to know the processor. A **translation table** stores the processor on which the data resides, its local address and the processor's memory. A distributed **translation table** turned out not to be sufficient enough, so PARTI uses a **paged translation table**. In brief, the **translation table** is decomposed into fixed-sized pages, and each page lists the home processor and offsets. Each processor maintains such a page table. This enables to access globally indexed **distributed array**s that are mapped onto processors in an irregular manner [43].

Since it is not possible in irregular patterns to predict at compile time what data must be prefetched, the original sequential loop is transformed into two constructs, namely the **inspector** and the **executor** [139]. The **inspector** loop examines the data references (made by a processor) and calculates what off-processor data needs to be fetched. The **executor** loop uses this information to implement the actual computation.

Each **inspector** produces a set of schedules, which specify the communication calls needed to either obtain copies of stored data or modify the contents of specified off-processor locations or accumulate values to specified off-processor memory locations. Hash tables are used to generate communication calls that transmit only a single copy of each processor datum.

Software ***caching*** can be used to reduce the volume of communication between processors. All data that is needed by a set of irregular references is prefetched. The same off-processor data may be accessed repeatedly, but only a single copy of that data must be fetched from off-processor. During the schedule generation process each processor sends the list of data it needs from all other processors, and it receives the list of data it must send to other processors [45]. Software Caching is further divided into Simple Software Caching and Incremental Software Caching [43].

PARTI also uses ***data reuse*** and **communication coalescing**. **Communication coa-**

lescing collects many data items for the same processor into a single message to reduce the number of message startups. PARTI defines three types of **communication coalescing** [43]: Simple communication aggregation, where all data each pair of processors needs to exchange is packed into a single message; communication vectorization and schedule merging.

Partial Redundancy Elimination (PRE)

abstract:

PRE is a technique for optimizing code by suppressing partially redundant computations, and is used in optimizing compilers for performing common subexpression eliminiation and strength reduction [2]. [2] describes an **Interprocedural Partial Redundancy Elimination algorithm (IPRE)** which is used for optimizing placement of communication statements and communication preprocessing statements in **distributed memory** compilations. In this environment the communication overhead can be decreased by message aggregation. In other words, each processor requests a small number of large amounts of data. The optimization is obtained by placing a preprocessing statement to determine the communicated data. The information is stored in a **communication-schedule**. The developed **IPRE** algorithms is applicable on arbitrary recursive programs [2].

aims:

code opimization

implementation platform:

Intel iPSC/860 hypercube

portability:

PRE uses a ***Fortran D*** compiler

related work:

other flow-sensitive interprocedural problems: concept of Super Graph [2]; FIAT for interprocedural analysis

application:

Euler Solver

people:

Gagan Agrawal, Joel Saltz, Raja Das

{gagan, saltz, raja}@cs.umd.edu

institution:

Department of Computer Science, and UMIACS, University of Maryland, College Park, MD 20742, USA

http://www.cs.umd.edu/projects/hpsl.html

key words:

code optimization, **distributed memory**

Partitioned In-core Model (PIM)

This is one of the three basic models of ***PASSION*** for accessing **out-of-core** arrays. It is a variation of the ***Global Placement Model***. An array is stored in a single global file and is logically divided into a number of partitions, each of which can fit in the main memory of all processors combined [143]. Hence, the computation problem is rather an in-core problem than an **out-of-core** one.

PASSION (Parallel And Scalable Software for Input-Output)

abstract:

PASSION is a runtime library that supports a ***loosely synchronous SPMD*** programming model of parallel computing [33]. It assumes a set of disks and **I/O node**s which can either be dedicated processors or some of the **compute node**s can also serve as **I/O node**s. Each of these processors may either share the set of disks or have its local disk. What is more, PASSION considers the ***I/O problem*** from a language and compiler point of view. ***Data parallel*** languages like ***HPF*** and **pC++** allow writing parallel programs independently of the underlying architecture. Such languages can only be used for ***Grand Challenge Applications*** if the compiler can automatically translate **out-of-core (OOC)** ***data parallel*** programs. In PASSION, an **OOC** ***HPF*** program can be translated to a **message passing** node program with explicit parallel I/O.

aims:

software support for high performance parallel I/O at the compiler, run-time and file system levels [33]

implementation platform:

Intel Delta, IBM SP2

related work:

Choudhary and Bordawekar also deal with **OOC** stencil problems in [13]

Panda: chunking for performance improvement, ***PIOUS***, ***MPI-IO, Vesta, PARTI, CHOAS, Vienna Fortran***, ***HPF, Bridge File System***, ***disk-directed I/O***

data access strategies:

Two-Phase Method

people:

Alok Choudhary, `chaudhar@ece.nwu.edu`

Rajesh Bordawekar, Rakesh Krishnaiyer, Micheal Harry, Ravi Ponnusamy, Tarvinder Pal Singh, Rajeev Thakur, Juan Miguel del Rosario, Sachin More, K. Sivaram, A. Dalia, Bhaven Avalani

{`rajeh, rakesh,mharry,ravi,tpsingh,thakur,mrosario`}`@cat.sys.edu` J. Ramanujam, `jxr@gate.ee.lsu.edu`

Ian Foster

institution:

ECE Department, Syracuse University, NY 13244, USA

ECE department, Luisiana State University, Baton Rouge, LA 70803, USA

ANL Northwest Pacific Architecture Center, 3-201 CST, Syracuse University

Northwestern University, USA

`http://ece.nwu.edu/ chaudhar/passion`

key words:

I/O library, **distributed memory**, **out-of-core** computation, ***SPMD***, compiler, ***Two-Phase Method***

details:

PASSION uses an interface above the existing ***parallel file system*** and improves abstractions to the underlying file system. The library can either be used directly by the application

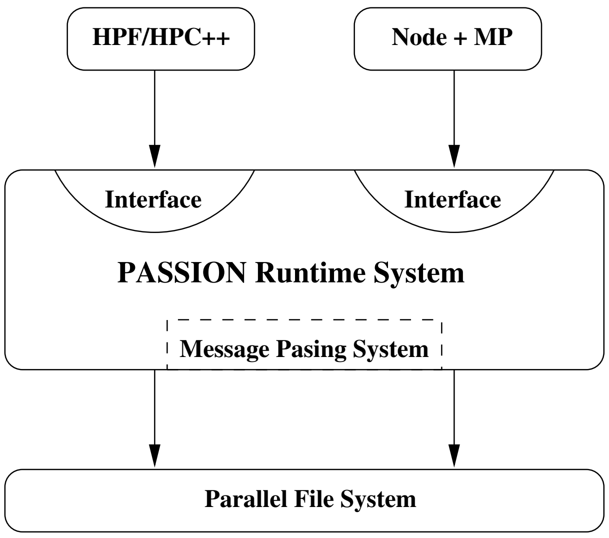


Figure 2.18: PASSION Software Architecture

programmer or by node programs translated by a compiler (see Figure 2.18).

1 Out-of-core Model

PASSION distinguishes between an in-core and an **out-of-core** program. Whereas in an in-core program the entire amount of data (e.g. elements of a **distributed array** in a **distributed memory machine**) fits in the local main memory of a processor, large programs and large data do not fit entirely in the main memory and have to be stored on disk. Such data arrays are referred to as **Out-of-core Local Array (OCLA)** [141]. Unfortunately, many massively parallel machines such as **CM-5**, ***Intel iPSC/860***, ***Intel Touchstone Delta*** or ***nCUBE***-2 do not support **virtual memory** otherwise the **OCLA** can be swapped in and out of disk automatically, and the ***HPF*** compiler could also be used for **OOC** programs.

The portion of a local array which is in main memory is called **In-Core Local Array (ICLA)** and all computations are performed there. Additionally, this part is stored in a separate file called **Local Array File (LAF)**. During computation parts of the **LAF** are fetched into the **ICLA**, new values are computed and the **ICLA** is stored back into appropriate locations in the **LAF**. The size is specified at compile time and depends on the available.

The larger the **ICLA** the better, as it reduces the number of disk accesses [141].

2 Compiler Design

OOC compilation techniques also require knowledge of in-core compilation, thus, in-core compilation is presented firstly.

2.1 In-core Compilation

An array assignment statement (see sample program ***High Performance Fortran***) is translated using the following steps [141]:

1. The ***distribution*** pattern is analyzed.
2. The type of required communication is detected.
3. Data partitioning is performed, and the lower and upper bound for each participating processor is calculated.
4. Temporary arrays are used if the same array is used in both RHS and LHS of the array expression.
5. The sequential **F77** code is generated.
6. Calls to runtime libraries are added to perform **collective communication**.

Communication between the processors is established in a ***SPMD*** (***loosely synchronous***) style where all processors have to synchronize before communication. It is also referred to as a **collective communication** [11].

2.2 Out-of-core Compilation

Here many factors have to be considered: data ***distribution*** and disks, number of disks, ***prefetching*** and ***caching*** techniques. However, some techniques from in-core compilation can also be employed (e.g. **stripmining** where loop iterations are partitioned so that data of fixed size can be operated on in each iteration [141]). The **LAF** of a processor is divided into **slab**s whose sizes are equal to the **ICLA**. This means that data partitioning consists of

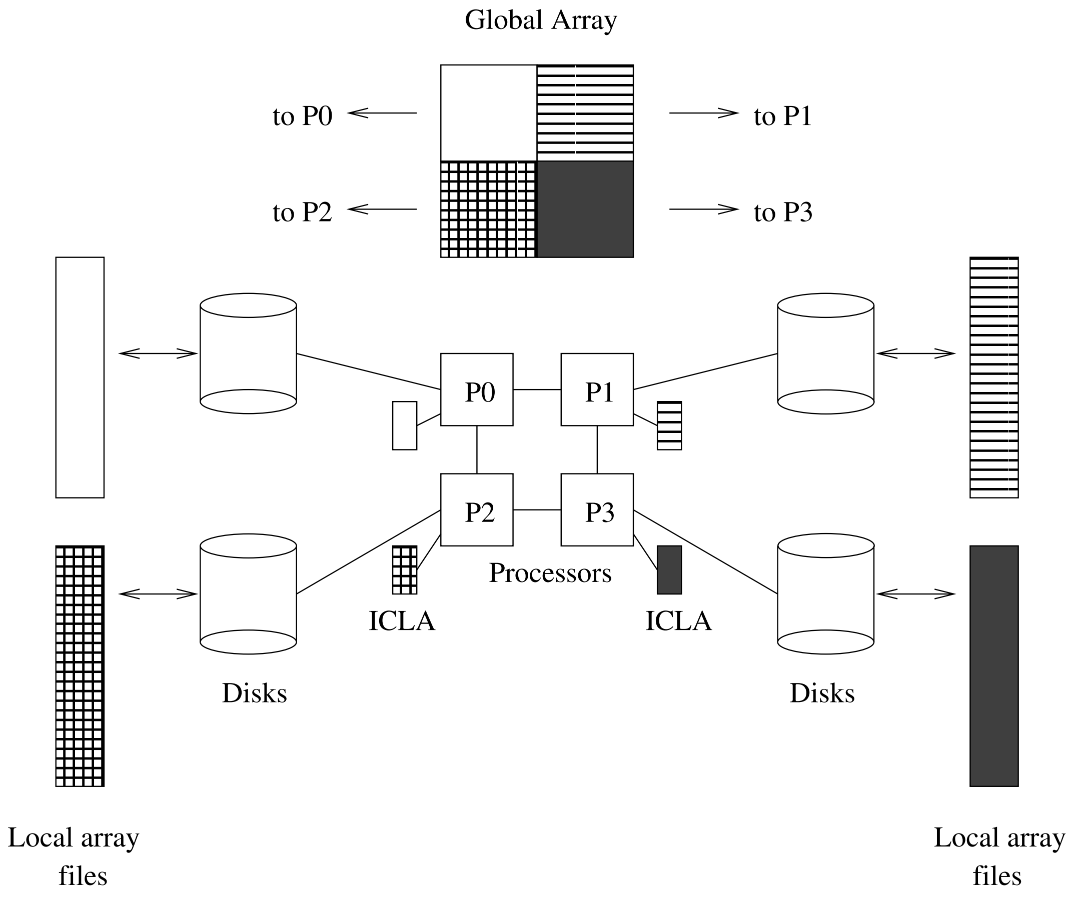


Figure 2.19: Model for Out-of-Core Compilation

two levels (see Figure 2.19).

Data is first partitioned among processors and then data within a processor is partitioned into **slab**s which fit in the processor's main memory. A sample array assignment can look as follows:

A(i, j) = (B(i-1, j) + B(i+1, j) + B(i, j-1) + B(i, j+1)) / 4

Since processors need data from neighboring processors in order to compute overlapping arrays, a form of communication between the processors is required, either an **Out-of-core Communication Method** or an **In-core Communication Method**.

- **OOC Communication Method**: The compiler has to determine what off-processor data is required for the entire **OOC** local array. Hence, no communication is required during the computation on each **slab**. After the computation, the **slab** is written back to disk. Communication is separated from computation. The communication stage also requires accessing data from other processors, and PASSION provides two options:

  1. **Direct File Access**: Any processor can directly access any disk. The processor directly reads data from the **LAF** of other processors. Drawbacks are greater disk contention and high granularity of data transfer.
  2. **Explicit Communication**: Each processor accesses only its own **LAF**. Data is read into memory and sent to other processors. Since the data to be communcatited has to be read from disk, there is no contention.

- **In-core Communication Method**: The compiler analyses each **slab** instead of the entire **OOC** array. Communication is not performed collectively but interleaved with computation on the **ICLA**. A special data structure called **Out-of-core Array Descriptor (OCAD)** is employed to store such information as array size, ***distribution*** or storage patterns.

3 Data Structures

PASSION provides support for reading/writing entire arrays as well as arrays stored in files. Following data structures are available [33]:

- **Out-of-core Array Descriptor (OCAD)**
- Parallel File Pointer (PFILE)
- Prefetch Descriptor (see ***data prefetching***)
- Reuse Descriptor (see ***data reuse***)
- Access Descriptor

4 Compiler Support

A compiler for programs with arrays that are too large to fit in main memory has to perform following two main tasks [32]:

- Generate run-time calls to perform read/write of the arrays.
- Perform automatic program transformation to improve I/O performance.

An **OOC** program is compiled in two phases:

- Phase I: Global Program Preprocessing
- Phase II: Local Program Preprocessing
  1. Work Distribution
  2. Loop Optimizations
  3. Local Dataflow Analysis
  4. Communication Optimizations
  5. I/O Optimizations
  6. Inter and Intra File Organizations

pC++ (parallel C++) see ***CHAOS***

PDS (PIOUS Data Server) see ***PIOUS***

PE (processing element) see ***SIMD***

peer-to-peer see ***Agent Tcl, SHORE, TOPs***

permutation

Permutation is important where data resides on disk, and it is central to the theory of I/O complexity. [41] examines the class of bit-matrix-multiply/complement (BMMC) ***permutation***s for parallel disk systems.

PFS (Parallel File System)

PFS is the file system for ***Intel Paragon***'s operating system ***OSF/1***. In general, ***OSF/1*** provides two forms of parallel I/O [74]:

- PFS gives high-speed access to a large amount of disk storage, and is optimized for simultaneous access by multiple nodes. Files can be accessed with parallel and non-parallel calls.

- Special I/O system calls, called parallel I/O calls, give applications better performance and more control over parallel file I/O. These calls are compatible with the ***Concurrent File System*** (CFS) for ***Intel iPSC/860 hypercube***.

OSF/1 can be executed with or without PFS. PFS internally consists of one or more stripe directories, which are the mount points for separate **UFSs (UNIX File Systems)**. Moreover, several file systems are collected together into a unit that behaves like a single large file system. File names and path names work in the same way as in **UFS**.

PFS performs I/O in parallel whenever possible, even if the user does not use parallel I/O calls. Parallelism can be obtained in two forms [74]:

- Different **stripe unit**s can be dealt with in parallel if a single node performs I/O on a block that is larger than one **stripe unit**.

- If two nodes read/write different file systems at the same time, the disk operation can proceed in parallel as well.

physical data locality see ***ViPIOS***
Physical Layer Storage Object (PSO) see ***HFS***
PIM see ***Partitioned In-core Model***

PIOFS (IBM AIX Parallel File System)
abstract:

PIOFS is a ***parallel file system*** for the ***IBM SP2***. It uses UNIX like read/write and logical partitioning of files. Furthermore, logical views can be specified (subfiles). PIOFS is capable of scaling I/O performance as the underlying machine scales in compute performance [74]. What is more, applications can be parallized in two different ways: logically or physically [74]. Physically means that a file's data is spread across multiple server nodes whereas logically refers to the partitioning of a file into subfiles. Other features: faster job performance, **scalability**, portability and application support, and file ***checkpointing***.

aims:

acessing data without overhead of maintaining multiple data files

implementation platform:

IBM SP2

data access strategies:

A file can both be treated as a normal file or it can be partitioned into subfiles where each subfile can be processed in parallel by a separate task [74].

portability:

PIOFS can coexist with other AIX Virtual File Systems in both storage and **compute node**s, and it complements existing file systems [74]. C and Fortran programs can use the parallelism.

related work:

Vesta

institution:

IBM Coperation, Department of PDQA, RS/6000 Division, Sommers, NY 10589, USA

key words:

file system, views, ***checkpointing***

PIOUS (Parallel Input-OUtput System)

abstract:

Since in ***metacomputing*** environments I/O facilities are not sufficient for a good performance, the virtual, ***parallel file system*** PIOUS was designed to incorporate true parallel I/O into existing ***metacomputing*** environments [107] without requiring modification to the target environment, i.e. PIOUS executes on top of a ***metacomputing*** environment. What is more, parallel applications become clients of the PIOUS task-parallel application via library routines. In other words, PIOUS supports parallel applications by providing coordinated access to file objects with guaranteed consistency semantics [109].

aims:

incorporate true parallel I/O into ***metacomputing*** environments

implementation platform:

Sun SS2 IPC workstations with SunOS 5.3, SunOS4.1.3/5.3-4, IRIX 4.0.5/5.3, ***OSF/1*** 2.1, HP-UX, AIX

data access strategies:

PIOUS allows three logical views of a file object: global, independent and segmented [107].

1. global: A file appears as a linear sequence of data bytes, and all processes in a group share a single file pointer .
2. independent: A file appears as a linear sequence of data bytes (local file pointers for each process).
3. A file appears in the natural segmented form (specified segments are accessed via local file pointers)

Views only define the way a file is accessed, but do not alter the physical representation. Moreover, all processes in a group have to open a file with the same view [107]. Between groups, a form of file locking has to be considered (see also ***MPI-IO***). However, PIOUS allows accessing a multi-segemented file, which could lead to some problems concerning the EOF (end-of-file).

portability:

Version 1 of PIOUS is implemented for the ***PVM metacomputing*** environment, hence, PIOUS is supposed to be installed effortlessly on any machine that ***PVM*** has been built on, provided that the system has a UNIX style interface (see ***I/O interfaces***) [109]. In general, PIOUS should be independent of the underlying hardware and software systems.

related work:

HiDIOS, Bridge File System, ***CFS***, ***nCUBE***'s ***parallel file system***, ***Vesta***

people:

Steven A. Moyer, V.S. Sunderam

`{moyer,vss}@mathcs.emory.edu`

institution:

Department of Mathematics and Computer Science, Emory University, Atlanta, GA 30322, USA

http://www.mathcs.emory.edu/Research/PIOUS.html

key words:

file system, **distributed memory**

example:

The following example in C from [109] accesses two files, one on the default set of data server hosts and the other on the set of hosts specified in the program text.

```
#inlcude <pious1.h>

main ()
{
  int fd;
  char buf [4096];
  struct pious_dsvec dsv[2];

  /* access file on default set of data server hosts */

  fd = pious_open ("file1.dat",
       PIOUS_RDWR | PIOUS_CREAT | PIOUS_TRUNC,
       PIOUS_IRUSR | PIOUS_IWUSR);
  pious_write (fd, buf, (pious_sizet)4096);
  pious_lseek (fd, (pious_offt)0, PIOUS_SEEK_SET);
  pious_read (fd, buf, (pious_sizet)4096);
  pious_close (fd);

  /* access file on hosts marcie and patty */

  dsv[0].hname = "marcie";
  dsv[0].spath = "/users/moyer/piousfiles";
  dsv[0].lpath = "/users/moyer/piouslogs";

  dsv[1].hname = "patty";
  dsv[1].spath = "/users/moyer/piousfiles";
  dsv[1].lpath = "/users/moyer/piouslogs";

  fd = pious_sopen (dsv, 2,
       "file2.dat",
       PIOUS_RDWR | PIOUS_CREAT | PIOUS_TRUNC,
        PIOUS_IRUSR | PIOUS_IWUSR);
  pious_write (fd, buf, (pious_sizet)4096);
  pious_lseek (fd, (pious_offt)0, PIOUS_SEEK_SET);
  pious_read (fd, buf, (pious_sizet)4096);
  pious_close (fd);
}
```

details:

[107] defines some principles that are essential to develop a scalable, high-performance network ***parallel file system***.

1. Transport and Native File System Independence: PIOUS must be easily portable to a variety of ***metacomputing*** environments implemented on different hardware platforms. Basic **message passing** and a UNIX style file system are sufficient transport and data access mechanisms.

2. Asynchronous Model of Operation: This model is used in order to achieve high performance. Additionally, it is supposed to be free of inherent **bottleneck**s [107] and it is facilitated by ***concurrency control***.

3. **Data Declustering** (***distribution*** of the file data): It allows accessing files in parallel, which results in a potential increase in transfer rate proportional to the number of devices.

4. Access Mechanism and Policy Independence

1 Software Architecture

An overview of the PIOUS software architecture is depicted in Figure 2.20. PIOUS consists of a **PIOUS service coordinator (PSC)**, a set of **PIOUS data servers (PDS)** and library routines linked with client processes [107]. Messages between client processes and components of the PIOUS architecture are exchanged by an underlying transport mechanism. Furthermore, permanent storage is accessed by data servers via a native file system [107]. In other words, a **PDS** corresponds to an **I/O daemon** [108].

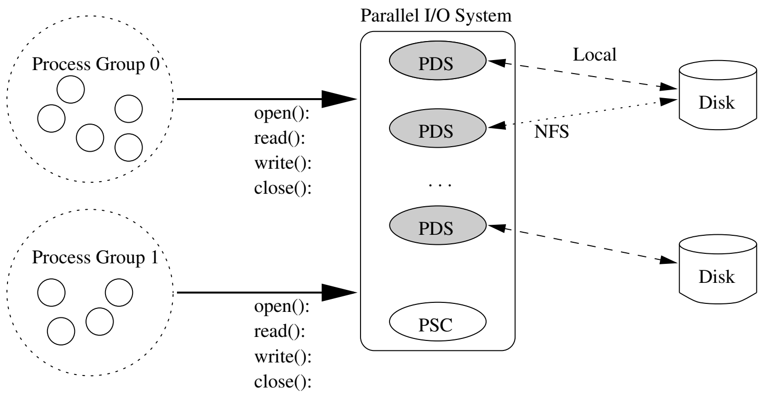


Figure 2.20: PIOUS Software Architecture

A single **PSC** initiates activities within the system and participates only in major system events rather than in general file access. In particular, it manages file metadata and system state. Thus, it does not represent the system **bottleneck** [107].

A **PDS** is located on each machine over which files are declustered. Moreover, **PDS**s are independent and do not communicate, i.e. each **PDS** accesses a file system local to the machine on which it resides, and two **PDS**s can share a file system via a network file service [107]. Each PIOUS client process is linked with library routines that translate file operations into **PSC/PDS** service requests [107]. Due to the independence of parallel data servers asynchronous operations are emphasized.

2 Transactions in PIOUS

PIOUS uses stable and volatile transactions, and both guarantee **serializability** of access (see also ***Strict Two-Phase Locking***).

- stable: "traditional" transaction, two-phase commit, synchronous disk write operations, guarantees that coherence is maintained (fault tolerance)
- volatile: "light-weight", does not guarantee fault tolerance, asynchronous (Read and write operations are implemented as volatile transactions [108].)

3 File Model and Interface

The current interface is rather similar to UNIX. In detail, PIOUS tries to use similar function arguments and behavior, but it diverges from UNIX by implementing two-dimensional file objects and sophisticated access coordination mechanisms [107]. Files are declustered by placing each data segment on a single **PDS** within a logical file system. If the number of data segments exceeds the maximum number of data servers, the segments are mapped in a **round-robin** fashion. For example, if a file consists of four segments and the number of **PDS** is 2, each of the **PDS**s gets two of the four file segments. **PDS**-0 receives segment 0 and 2 whereas **PDS**-1 is responsible for the segments 1 and 3.

PIOUS Data Server (PDS) see ***PIOUS***

PIOUS Service Coordinator (PSC) see ***PIOUS***

PN (processing node) see ***ParFiSys (Parallel File System)***

point-to-point communication see ***MPI***

Portable Parallel File System (PPFS)

abstract:

PPFS is a file system designed for experimenting with I/O performance of parallel scientific applications that use a traditional UNIX file system or a vendor-specific ***parallel file system*** [50]. PPFS is implemented as a user level I/O-library in order to obtain more experimental flexibility. In particular, it is a library between the application and a vendor's basic system software. Furthermore, the correct usage of PPFS requires some assumptions [51]: The underlying file system has to be a standard UNIX file system, which allows the file system to be portable across a wide range of UNIX systems without changing the kernel or the device drivers. Additionally, PPFS has to sit on top of a **distributed memory** parallel machine. It is assumed that applications are based on a **distributed memory message passing** model.

aims:

- identify the major issues for performance in parallel I/O (solve the ***I/O problem***)
- A flexible ***API*** should serve as a tool for specifying access pattern hints and

controling the file system behavior.

- PPFS should be used for exploring data ***distribution***, distributed ***caching*** and ***prefetching*** techniques [71][73].
- exploration of distributed techniques for dynamically classifying file access patterns

implementation platform:

Intel Paragon, network of UNIX workstations, Thinking Machines **CM-5**,

data access strategies:

PPFS uses parallel files which consist of a sequence of records, which in turn is the unit of access. Additionally, PPFS defines a set of access modes that have to be specified when opening or creating a parallel file. Although a file can be accessed in parallel, this does not imply that **serializability** can be dropped when using a ***shared file pointer***. Asynchronous calls allow PPFS applications to overlap computation with I/O operations [71]. PPFS defines several parallel access patterns [72]:

- **Direct Access**: The data spread out over several disks may be accessed in parallel by clients.
- **Strided Access**: A strided file can be read by clients record after record with a stride `s`, i.e. the next record after `i` is `i+s`. This pattern can lead to a disjoint access of the whole file.
- **Synchronized Access**: In the **strided access** each client is acting independently. In contrast, the synchronized access requires a synchronization of the clients. Problems like **Producer-Consumer** or barrier synchronization can be dealt with.
- **Ordered Access**: The logical records of a parallel file do not necessarily correspond to any logical order that implies any sequential access.
- **Overlapping** or **Disjoint Access**: The file access of clients may overlap or they can access disjoint portions of a file.

portability:

portable across a number of parallel platforms [73] with the following requirements: underlying UNIX file system and a typed **message passing** library

related work:

PIOUS, ***PASSION***, ***CFS***, ***Vesta***, ***PFS***, ***disk-directed I/O***

people:

Daniel A. Reed, Andrew A. Chien, Chris Elford, Chris Huszmaul, Jay Huber, Tara Madhyastha, Daiv S. Blumenthal, James V. Huber Jr.

{reed, achien, elford}@cs.uiuc.edu

institution:

Department of Computer Science, University of Illinois, Urbana, IL 61801

http://www-pablo.cs.uiuc.edu/Projects/PPFS/ppfs.html

key words:

file system, **client-server**, ***caching***, **distributed memory**, **message passing**

example:

The following program from [71] reads a file:

```
#define CACHE_SIZE 512*1024
#define CACHE_BINS 29
#define READ_SIZE 4096
#define READ_COUNT 16384

int readPtr, clientCount, lowClient;
char fileBuffer [READ_SIZE];

ppfs_client_cache (CACHE_SIZE, CACHE_BINS);

if (mynode () == ppfs_cient0())
{
  readPtr = ppfs_open(Read.Example", pf_in);
  clientCount = ppfs_clients();
  lowClient = ppfs_client0();

  for (i = lowClient; i < clientCount+lowClient; i++)
    ppfs_send_file (readPtr, mytype (), i);
}
else
  readPtr = ppfs_recv_file ();
```

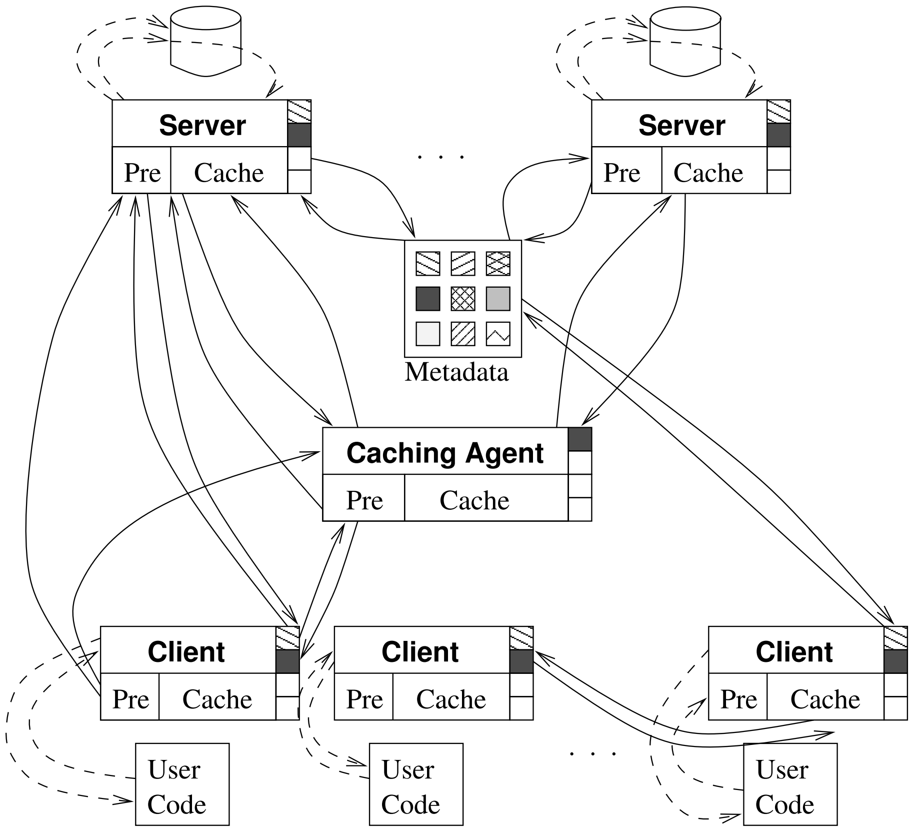


Figure 2.21: PPFS Design

```
for (recordNumber=0; recordNumber>READ_COUNT; recordNumber++)
  read_logical (readPtr, fileBuffer, recordNumber, 1);

ppfs_close (readPtr);
ppfs_exit (0);
```

details:

The design is based on the **client-server** model. Figure 2.21 illustrates the objects (rectangles) and the interactions in PPFS (dashed arrows represent system calls, and solid arrows refer to interprocess communication or **message passing** [50]). **I/O node**s (servers) manage I/O devices and handle requests from clients which are represented by **application process**es. **I/O server**s use standard UNIX files to store data, but they do not know anything about the contents of the files they manage [51]. Additionally, requests can have a priority argument in order to re-order the requests in the queue. Since ***prefetching*** is low level, it can also be controlled by the application.

A user application is linked with a PPFS library and makes calls to this library in order to perform I/O operations [72]. The usercode is transformed into a PPFS client, which consists of user code, a client cache, a prefetch unit, bins for locating information or metadata, messaging support, and client control routines. Each server program is executed by an **I/O node**. Such a server consists of a cache, a prefetch unit, bins for holding open file **meta-data**, server messaging support and an underlying UNIX file system. In addition to servers and clients, PPFS specifies intermediate objects: the **meta-data** server and a **cache agent**.

Clients have the task to access the file system and to maintain **meta-data** of a parallel file. **Meta-data** contains additional information which is necessary for accessing distributed files and how the file is organized (e.g. arrangement of file striping or location of the file). **Meta-data** is analogous to UNIX **i-node**s and open file pointers. A **meta-data** server communicates with **I/O server**s to inform them about the status of the files.

Caching can occur at several levels within PPFS. Each **I/O node** (**I/O server**) maintains a block cache of the data on its disk, and each client also has its own cache. What is more, a **caching agent** that performs application level ***caching*** and ***prefetching*** is available. A shared file can be treated as a **producer/consumer** resource. A read request for this shared file will trigger an **agent** request. See also ***Agent Tcl***.

POSIX see ***ParFiSys (Parallel File System), Cache Coherent File System (CCFS)***
PRE see ***Partial Redundancy Elimination***
prefetching see ***data prefetching***
proactive disk and file buffer management see ***SPFS***
processing element (PE) see ***SIMD***
processing nodes (PN) see ***ParFiSys (Parallel File System)***
producer/consumer see ***Portable Parallel File System (PPFS)***
producer-driven see ***in-core communication***

programming language E

E, a variant of C++, is a persistent programming language originally designed to ease the implementation of data-intensive software systems, e.g. database management systems, requiring access to huge amounts of persistent data [150]. E uses an interpreter called **E persistent Virtual Machine** (**EPVM**). The interpreter is used to coordinate access to persistent data that is stored using the **EXODUS Storage Manager (ESM)**.

PSC (PIOUS Service Coordinator) see ***PIOUS***
PSO (Physical layer Storage Object) see ***HFS***
PSS (ParSet Sever) see ***SHORE***
public domain see ***Agent Tcl***

PVM (Parallel Virtual Machine)
abstract:

PVM is a software tool allowing a heterogeneous collection of workstations and supercomputers to function as a single high-performance parallel machine [55], i.e. a workstation cluster can be viewed as a single parallel machine (see also ***metacomputing***). PVM can be used in both parallel and ***distributed computing*** environments. A **message passing** model is used to exploit ***distributed computing*** across the array of processes or processors. Moreover, data conversion and task scheduling are also handled across the network.

aims:

PVM should link computing resources. What is more, the parallel platform can also consist of different computers on different locations (heterogeneity). PVM makes a collection of computers appear as a large **virtual machine** [55]. The principles upon which PVM is based are [55]: user-configured host pool, translucent access to hardware, process-based computation, explicit **message passing** model, heterogeneity support and multiprocessor support.

implementation platform:

PVM can be installed on workstation clusters as well as on highly parallel systems such as ***Intel Paragon***, Cray T3D and Thinking Machine **CM-5**. CRAY and DEC have created PVM ports for their T3D and DEC 2100 **shared memory** multiprocessors [55].

data access strategies:

PVM provides communication constructs for sending and receiving data, and high-level primitives such as broadcast, barrier synchronization and global sum (as an example for a reduction operation). Tasks can exchange messages with no limitation concerning the size or the number of messages [55]. PVM supports asynchronous blocking send/receive and nonblocking receive functions. A synchronous communication is possible as well. The message order is guaranteed to be preserved. Unlike ***MPI*** sending of a message comprises three steps. First, a buffer has to be initialized, second, the message has to be packed and, finally, the message has to be sent. The communication is based on TCP, UDP and UNIX-domain sockets.

portability:

PVM is portable and runs on a wide variety of platforms [55]. The libraries of PVM are written in C. ***PIOUS*** is designed for PVM.

related work:

MPI, ***p4***, ***Local File Server***, ***Express***

people:

Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, Vaidy Sunderam

institution:

Oak Ridge National Laboratory (the prototype system PVM 1.0 was constructed there by Vaidy Sunderam and Al Geist)

University of Tennesse (PVM 2.0 was written there)

Carnegie Mellon University

`http://www.netlib.org/pvm3/book/pvm-book.html`

key words:

I/O interface, **message passing**, ***MIMD***, **virtual machine** example:

1 Master-Slave Approach

The technique to underly the first two code fragments is called **master-slave** and is one example of data parallelism [136]. Thus, one master process prompts the slave processes to handle the data which they receive. A **master-slave** program in PVM consists of two separate programs, rather than a single one which is common in other parallel programming languages, e.g. ***MPI***. Before the data can be exchanged, the master process has to spawn several slave processes. The following two code fragments from [136] show the master and the slave code for adding two vectors:
Master process:

```
#include <stdio.h>
#include "pvm3.h"

main()
{
  int successfully_spawned,
  tids[5],
  i,j,
  nproc,current_process,
  x[5],y[5];
  ...


  nproc=5;

  // generate two vectors

  // spawn nproc processes
  successfully_spawned = pvm_spawn("vector_sum_s", (char**)0, 0, "",
    nproc, tids);

  // check the spawning process
  if (successfully_spawned<nproc)
  {
    // kill processes which were not successfully spawned
    for( i=0 ; i<successfully_spawned ; i++ )
      pvm_kill( tids[i] );

    pvm_exit();
    exit();
  }

  // send different values to different processes
  for (i=0; i<nproc; i++)
  {
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&x[i],1,1);
    pvm_pkint(&y[i],1,1);
    pvm_send(tids[i],1);
  }


  // receive results from slave processes
  printf("\n\nThe processes computed following results:");
  for (i=0; i<nproc; i++)
  {
    pvm_recv(-1,-1);
    pvm_upkint(&current_process,1,1);
    pvm_upkint(&s[i],1,1);
  }

  // print resulting vector

  pvm_exit();
  exit(0);
}
```

Slave process:

```
  // receive data from master process
  pvm_recv(-1,-1);
  pvm_upkint(&x,1,1);
  pvm_upkint(&y,1,1);

  s=x+y;

  printf("I'm a slave");
  // send task id and computational result to master process
  pvm_initsend(PvmDataDefault);
  pvm_pkint(&mytid,1,1);
  pvm_pkint(&s,1,1);
  pvm_send(ptid, 1);

  pvm_exit();
  exit(0);
```

2 Single Program Multiple Data

PVM offers another possibility to write parallel programs and makes use of an ***SPMD*** programming style. Unlike the **master-slave** programs, these are not split up into two parts, but consist of only one single program code. In general, the communication between different processes is organized in the same way. However, processes which handle a mutual task are mostly grouped together.

```
n_proc=n_rows-1;

if( (my_gid = pvm_joingroup(MGROUP)) < 0 )
{
  pvm_perror( "Could not join group \n" );
  pvm_exit();
  exit( -1 );
}

// if I'm the first group member then spawn other processes
if (my_gid == 0)
  successfully_spawned=pvm_spawn("multiply_vector", (char**)0,
  PvmTaskDefault, (char*)0,n_proc,tids);

// sync on data receiving
pvm_barrier(MGROUP,n_proc+1);

// compute

// barrier after computation
pvm_barrier(MGROUP,n_proc+1);

// output after the multiplication procedure

pvm_barrier(MGROUP,n_proc+1);
pvm_lvgroup(MGROUP);
pvm_exit();
exit(0);
```

details:

1 PVM

PVM consists of two parts: the first part is a daemon, called **pvmd**, that resides on all computers to make up the **virtual machine**. Once PVM is started, a **pvmd** runs on each host of the **virtual machine**. Its tasks are message routing, process control and fault detection. A form of master and slave **pvmd** can be created, where only one master **pvmd** is permitted to configure the **virtual machine** such as adding and deleting hosts. However, if the master **pvmd** crashes, the whole **virtual machine** crashes, too.

The second part of PVM is a library of PVM interface routines. A user can access PVM resources in form of a collection of tasks, .i.e. an application has to consist of several co-operating tasks. Such tasks can be initialized and terminated across the network. What is more, the tasks can communicate with each other, can start or stop another task or add/delete computers form the **virtual machine** [55]. Such tasks are identified by a task identifier (TID) which has to be unique across the entire **virtual machine** and is involved in the exchange of messages between tasks. The task id is similar to the UNIX process id (PID).

PVM supports group functions where several tasks can be joined to one group. In brief, one task can join/leave a group at any time even without informing the other tasks of the corresponding group.

The programming library can be regarded as the interface between tasks and **pvmd**s. Thus, functions for message handling and service requests can be performed.

2 Comparison of PVM and MPI

[136] discusses the differences between PVM and ***MPI***. ***MPI*** is said to be faster on **MPP (Massively Parallel Processor)** hosts than PVM [56].

PVM goes one step further in portability than ***MPI***. Rather than running applications only on various **MPP**s which can be regarded as a single architecture, PVM programs can be compiled and executed on any set of different architectures at the time [56] so that a PVM executable can communicate with all the other executables residing an different machines. This is referred to as interoperability. The reason why ***MPI*** does not provide that feature is that the destination address of every message must be checked in order to determine whether it resides on the same host or on another one. Latter case could require data conversion due to different architectures. Here is the major difference in the development of the two approaches. The PVM project puts its emphasis on heterogenety which allows much more flexibility whereas ***MPI*** focuses on performance and uses native hardware in order to make **message passing** faster. However, even PVM uses native hardware for the communication among hosts of the same architecture. The loss of performance in favor of flexibility is due to the communication between hosts of different architectures where PVM uses the slower standard network communication functions.

Besides this, PVM allows C programs to send messages to Fortran programs and vice versa. Since C and Fortran support different language features, the ***MPI*** standard does not provide this kind of language interoperability.

Another advantage of PVM over ***MPI*** is the concept of the **virtual machine** [56] which allows process control and resource control. This means that tasks can be traced back such as finding out which task is running on which platform. Moreover, the resource manager allows dynamic configuration of the host pool which can either be done from the PVM console or within the application programs. All those features of creating an arbitrary collection of machines, which are treated as uniform computational nodes, are not supported by the ***MPI*** standard.

A further important issue is fault tolerance. Since large scale computer applications sometimes run over a long period of time, a crash of the underlying system would waste many hours of computation. Thus, PVM provides features for the notification of changes in the **virtual machine** or task failures [56]. In short, if any task fails, all the other effected tasks get a notify message instead of the expected one. If a new host is added to the **virtual machine**, another notification message is sent which can be used for balancing the workload among the new resources. Since tasks are considered to be static in ***MPI***, mechanisms for fault tolerance are not part of the standard.

However, an important feature makes the communication of ***MPI*** more flexible than that of the counterpart, namely the concept of **communicator**s. Although message tags and sender ids can be used to distinguish messages for different purposes, this approach does not savely enough distinguish between library messages and user messages. Thus, **communicator**s are a suitable feature for modular programming and information hiding by means of binding a communication context to a group. Finally, one of the most cited advantages of ***MPI*** over PVM is the non-blocking receive.

To sum it up, ***MPI*** is supposed to yield better performance on a single **MPP** due to its richer set of communication functions. On the other hand, PVM's power is the **virtual machine** which guarantees interoperabiltiy between different applications and fault tolerance on clusters of workstations and **MPP**s.

pvmd (PVM daemon) see ***PVM***

R

RAID (Redundant Array of Inexpensive Disks)

abstract:

RAID (Redundant Array of Inexpensive Disks - due to the destructiveness of the term "inexpensive", RAID is also known as Redundant Array of Independent Disks) organizes multiple independent disks into a large, high-performance logical disk, stripes data across multiple disks and accesses them in parallel to achieve high data transfer and higher I/O rates. What is more, disk arrays increase secondary storage throughput [58]. However, these large disk arrays have also a major drawback: they are highly vulnerable to disk failures [29]. An array with x disks is x-times more likely to fail. A solution to this problem is to employ a redundant disk array and error-correcting codes to tolerate disk failures. Even this model has a disadvantage: all write operations have to update the redundant information, which reduces the performance of writes in the disk array.

Another drawback of a RAID system is that the throughput is decreased for small writes. What is more, such small data requests are especially important for on-line transaction processing (see also ***SPFS***). Thus, a powerful technique called **parity logging** is proposed by [137] for overcoming this problem.

aims:

striping to improve performance, and redundancy to improve reliability

people:

David A. Patterson, Katherine Yelick, Ethan L. Miller, Peter M. Chen, Edard K.Lee, Garth A. Gibson, Randy H. Katz, John M. Hartmann, Ann L. Chervenak Drapeau, Lisa Hellerstein, Richard M. Karp, Arvind Krishnamurthy, Steven Lumetta, David. E. Culler

`{patterson, yelick}@cs.berkeley.edu`

institution:

Computer Science Division, Department of Electrical Engineering, University of California, Berkeley, CA 94720, USA

DEC Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301

School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3891, USA

Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60208, USA

`ftp://ftp.cs.berkeley.edu/UCB/raid/papers`

key words:

disk arrays, storage, striping, redundancy, error correcting codes

details:

1 Basics of disk arrays

Two orthogonal concepts are employed: **data striping** for improved performance and redundancy for improved reliability [29]. **Data striping** means ***distribution*** of data over multiple disks to make them appear as a single fast, large disk. Moreover, it improves aggregate I/O performance by servicing multiple I/O requests in parallel. In order to distinguish the organization of RAIDs, one has to consider the granularity of data interleaving and the method and pattern in which the redundant information is computed and distributed across the disk array. Another two facts have to be kept in mind: What method to select for computing the redundant information and what method to apply for distributing the redundant information across the disks. Most RAIDs use parity, but some also use **Hamming codes** or **Reed-Solomon codes** for calculating redundant information.

2 RAID levels

RAID appears in different levels which correspond to a specific technique of striping the data and computing the redundant information. Normally, different levels of RAIDs have different numbers (from 0 to 6), but also English names are used for their classification. Figure 2.22 illustrates the seven different RAID levels [29]:

- **Non-Redundant (RAID Level 0)**

  Since this level does not employ any redundancy at all, it has the lowest cost of any redundancy scheme. Thus, it offers the best write performance. However, the disadvantage of omitting redundancy is that any single disk failure will result in data-loss. RAIDs 0 are applied in supercomputing environments where performance and capacity are the primary concerns, rather than reliability.

- **Mirrored (RAID Level 1)**

  This level is also called **shadowing** or **mirroring** and uses twice as many disks as a non-redundant disk array. There are always two copies of the information, consequently, whenever data is written to disk, a copy has to be sent to the redundant disk array. The main advantage is that if data is lost, the second copy can be used. This is a typical application of a database where availability and transaction rates are more important than storage efficiency [29].

- **Memory-Style ECC (RAID Level 2, Hamming-coded)**

  If memory systems fail, their recovery costs much less than **mirroring** by using **Ham-**

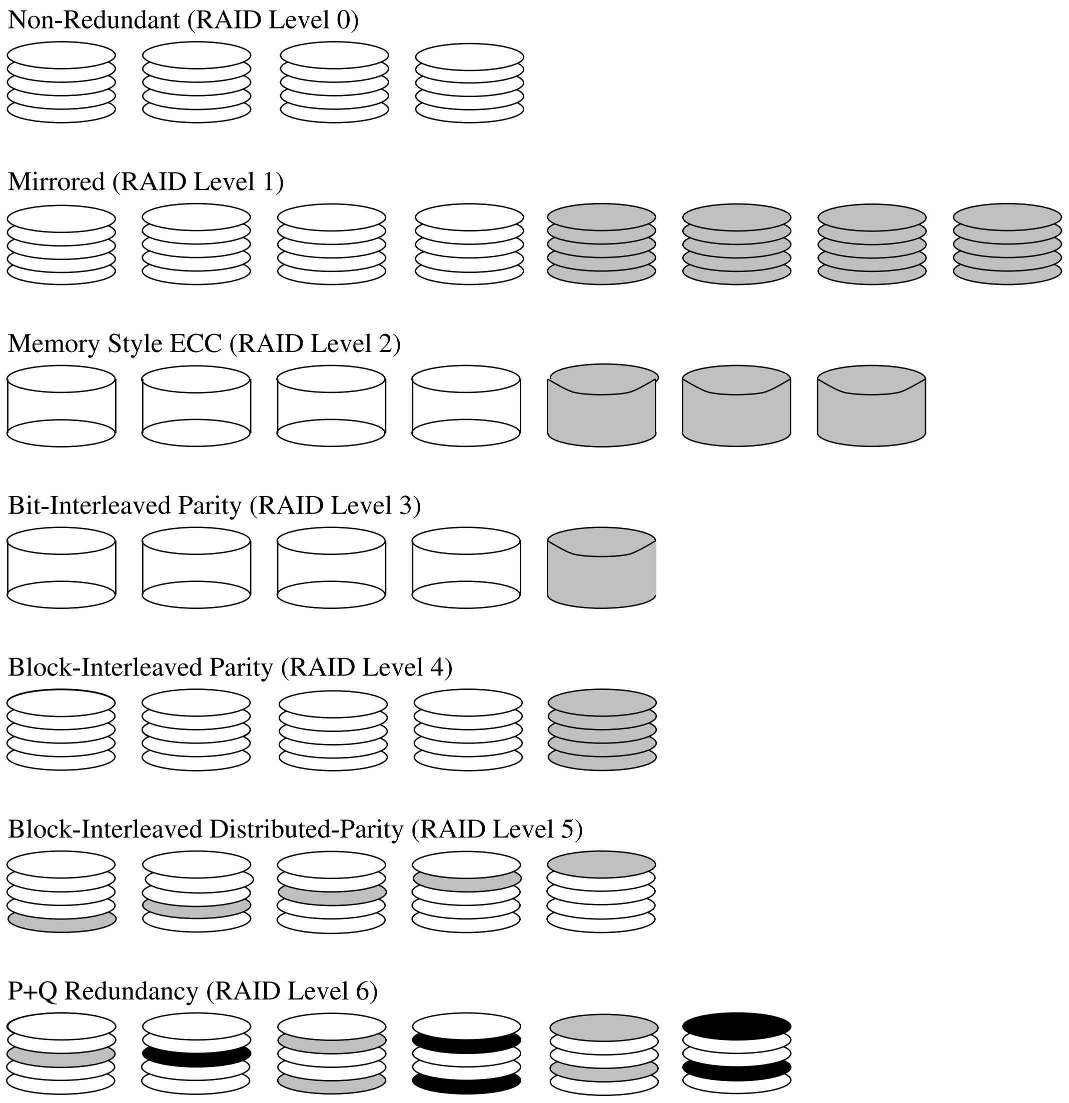

Non-Redundant (RAID Level 0)
Mirrored (RAID Level 1)
Memory Style ECC (RAID Level 2)
Bit-Interleaved Parity (RAID Level 3)
Block-Interleaved Parity (RAID Level 4)
Block-Interleaved Distributed-Parity (RAID Level 5)
P+Q Redundancy (RAID Level 6)

Figure 2.22: RAID Levels

ming codes. The number of redundant disks is proportional to the log of the total amount of disks, which results in an increase in the storage efficiency as the number of data disks increases [29]. Although multiple redundant disks are required, only one is needed to recover the lost information.

- **Bit-Interleaved Parity (RAID Level 3)**

  Here data is conceptually interleaved bit-wise over the data disks, and only one redundant disk is added to tolerate single disk failures. Each read and each write request accesses all disks. A write request also accesses the parity disk, i.e. only one request can be serviced at a time. Level 3 is used for applications that require high bandwidth rather than high I/O rates [29]. Moreover, their implementation is supposed to be simple.

- **Block-Interleaved Parity (RAID Level 4)**

  This level is similar to the bit-interleaved one, except that data is interleaved across disks of arbitrary size instead of bits. The size of these blocks is called **striping unit**, which is important for the execution of requests. For instance, requests smaller than the **striping unit** access only a single disk. There is only one parity disk which has to be updated on all write operations. Thus, the parity disk is the possible **bottleneck** in the RAID system [29]. Hence, the next level, block-interleaved distributed-parity is preferred to this level 4 solution.

- **Block-Interleaved Distributed-Parity (RAID Level 5)**

  The **bottleneck** of level 4 is eliminated by distributing the parity over all the disks available. Another advantage is that data is distributed over all disks rather than just over all but one. Parity **declustering** is a variant that reduces the performance degradations of on-line failure recovery [59].

- **P+Q Redundancy (RAID Level 6)**

  Parity can be considered as an error-correcting code that detects and corrects only single bit errors. Since disks became larger, codes which can correct multiple-bit errors are desirable, especially when applications require more reliability. **P+Q redundancy** uses **Reed-Solomon codes** to protect against up to two disk failures by using two redundant disks [29].

3 Comparison of the different levels

The three most important features of a RAID are reliability, performance and cost, but reliability is supposed to be the main reason for the popularity of disk arrays [29]. Reliability can be measured in terms of **mean-time-to-failure (MTTF)**. A system crash (power failure, operator error, hardware breakdown or software crash) can interrupt such I/O operations as writes, resulting in states where data is updated and parity not or vice versa. Although special techniques like hardware or power supplies can decrease the frequency of such crashes, no technique can prevent crashes 100%. In order to avoid the loss of parity by system crashes, information has to be recovered by non-volatile storages.

4 Stale data

It is important that some piece of information indicating the validity of a disk is stored, i.e. a variable has to indicate whether a redundant disk is valid or not. Some restrictions have to be considered:

- The invalid sectors of a disk have to be marked as invalid before any request that would normally access the failed disk can be serviced.
- On the other hand, a logically reconstructed sector has to be marked valid again before any write request that would normally write to the failed disk can be served.

If the first condition is violated, it would be possible to read stale data that is considered to have failed but works intermittently. A violation of the second condition would result in write operations that would fail to update the newly reconstructed sector.

An Analytic Performance Model is described which is different from the previous one for the reason that a closed queuing model with a fixed number of processes is used. Previous models have used open queuing models with Poisson arrivals. The closed model is supposed to model the synchronous I/O behavior of scientific, time-sharing and distributed systems more accurately [101].

5 Parity

Parity bits can be computed in many ways, but some requirements have to be satisfied:

- Stripe units belonging to the same **parity stripe** should not map to the same column. This is referred to as the "orthogonal RAID" property and guarantees that the failure of a single column does not result in data unavailability [102].
- In a RAID with n **stripe unit**s per **parity stripe**, the ith **parity stripe** unit should correspond to logical **stripe unit** j such that j div n = i. This guarantees that the parity for any write request that is aligned on a **parity stripe**, and a **parity stripe** in size can be computed by using only the data being written without reading old data [102].

The reason why small requests reduce the RAID performance stems from the disk access which can be broken into three components: seek time, rotational position time, and data transfer time. As for small writes, the time spent on seeking and positioning is greater than the actual data transfer time. **Parity logging** is a modification to RAID level 5. To start off, RAID level 4 is augmented with one additional disk, the log disk [137], which is initially considered as empty. Parity updates are buffered until they can be written to a log efficiently, i.e. the reintegration into a redundant disk array's parity is delayed until there are enough parity updates in the log. Since level 4 has a major disadvantage (see above), the log as well as the parity disk information are distributed. Each disk is further divided into manageable-sized regions.

RAID-I

abstract:

RAID-I ("RAID the First") is a prototype ***RAID*** level 5 system. It was designed to test workstation based-**file server**s concerning high bandwidth and high I/O rates. It is based on **Sun 4/280** workstations with 128 MB RAM and 28 5 1/4 inch SCSI disks and four dual-string SCSI controllers [99]. The most serious reason why RAID-I was ill-suited for high-bandwidth I/O was the memory contention.

institution:

Computer Science Division, Department of Electrical Engineering, University of California, Berkeley, CA 94720, USA

details:

1 General Information

A logical to physical mapping as it is used in ***RAID***s is defined by a pair of functions: the data-mapping-function and the parity-placement-function. The former is a one-to-one function that maps a logical block address to a physical address whereas the latter is a many-to-one function that maps a logical block address to a physical address [99].

RAIDs use some special terms for data mapping [99]:

- block: minimum unit of data transfer to or from a ***RAID*** device
- **stripe unit**: groups logically contiguous blocks that are placed consecutively on a single disk before placing blocks on a different disk
- **parity stripe**: Normally, stripe refers to a **parity stripe**, whereas a data stripe refers to a collection of logically contiguous units over which data is striped.
- row: minimal group of disks over which a **parity stripe** can be placed
- **small request**: fits entirely within a **stripe unit** (single disk)
- **moderate request**: spans multiple **stripe unit**s but does not use each disk more than once
- **large request**: large enough to use several disks more than once

2 I/O Request Servicing

Basically, the service of an I/O request is broken up into the following steps:

1. The I/O request is broken up into stripe requests, where each stripe is processed independently.

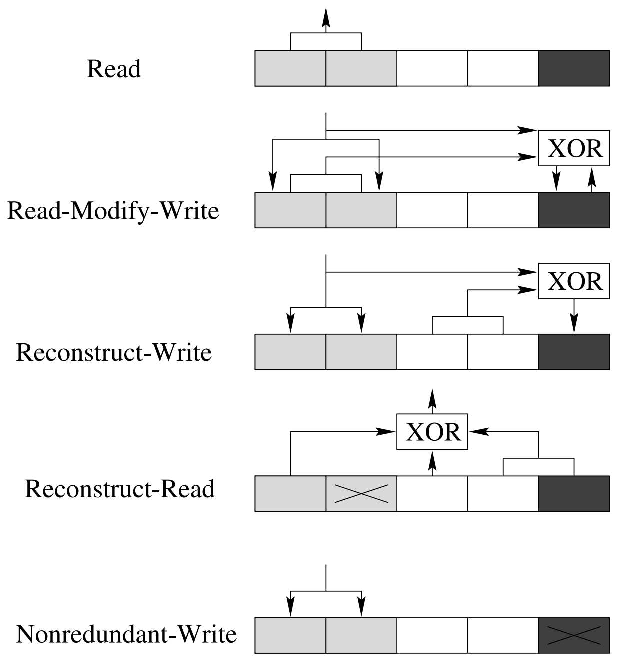


Figure 2.23: I/O Methods in RAID-I

2. The required stripes are locked in order to guarantee the consistency of the parity information associated with each stripe.

3. For each stripe an appropriate I/O method is chosen.

4. The stripe is unlocked afterwards.

An I/O method can be one of the following five. See also Figure 2.23. Before an I/O method can be selected, the validity of a chosen block has to be checked. This is done by consulting the validity of a disk block.

- Read: This method reads requested blocks only if all the blocks that are required are valid. If the physical request fails, the read method invokes the read-construct method.

- Read-Modify-Write: It is used if a relatively small portion of stripe is written. Here the priority is updated incrementally by **xoring** the new data, old data and parity. This is done in three steps:

    1. Read old data and old parity.

2. Compute new parity.

3. Write new data and new parity.

A single physical request failure invokes the reconstruct-write method.

- Reconstruct-Write: This method is invoked if a relatively large amount of a stripe is written. The three steps are similar to the one in read-modify-write except the data is read from the rest of the stripe in the first step. Similarly, a single physical failure causes the read-modify-write method to be invoked. More failures cause the stripe request to fail.

- Reconstruct-Read: It is used if one of the requested blocks is invalid. The parity serves to compute the contents of the invalid block.

- Nonredundant-Write: It is invoked if the blocks containing the parity are invalid. The parity does not need to be updated, but writing the data again is sufficient [99].

RAID-II

abstract:

RAID-II ("RAID the second") is a scalable high-bandwidth network **file server** and is designed for heterogeneous computing environments of diskless computers, visualization workstations, multimedia platforms and UNIX workstations [100]. It should support the research and development of storage architectures and file systems. It is supposed to run under **LFS**, the **Log-Structured File System**, developed by the Sprite operating system group at Berkeley [100]. What is more, **LFS** is specially optimized to serve as a high-bandwidth I/O and crash recovery file system.

institution:

Computer Science Division, Department of Electrical Engineering, University of California, Berkeley, CA 94720, USA

details:

A high-bandwidth file service is provided by a mainframe computer, and workstations are not likely to support high-bandwidth I/O in the near future [99]. RAID-II is different to the conventional workstation-based network **file server**s (see Figure 2.24), since it uses the

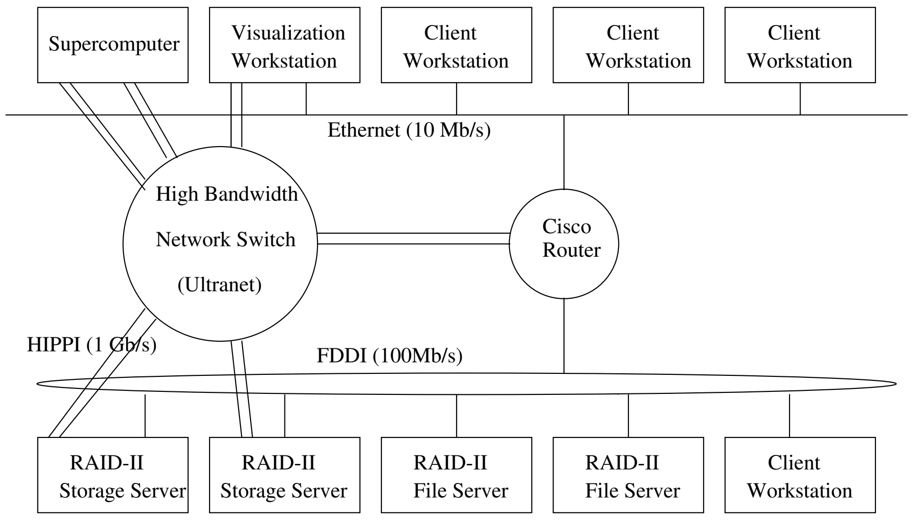


Figure 2.24: RAID-II Storage Architecture

network as the primary system backplane. It does not connect the high-bandwidth secondary storage system to the high-bandwidth network via a low-bandwidth bus, but connects it directly to the high-bandwidth network. Furthermore, the CPU is also connected to the network. Consequently, RAID-II is supposed to be more scalable than conventional network **file server**s [100].

RAID-II separates the storage system into **storage server**s and **file server**s. The **storage server** corresponds to the secondary storage system of conventional network servers. This concept has many advantages, e.g. if additional I/O bandwidth is required, it is possible to simply add more **storage server**s without changing the number of **file server**s. On the other hand, if **file server**s are overutilized, **file server**s can also be added. Another advantage is that redundancy schemes can easier be implemented, since file and **storage server**s are separated. However, the overhead to access the storage system is increased [100].

RAIDframe see ***SPFS***

raidPerf

The ***RAID*** Performance Measurement Tool, raidPerf, developed at Berkeley (University of

California, USA) is a generic tool for measuring the performance of concurrent I/O systems which can service multiple I/O requests simultaneously. The interface is similar to the one used in ***raidSim*** [103].

raidSim

A ***RAID*** simulator, raidSim, developed at Berkeley (University of California, USA) is an event-driven simulator for both modeling non-redundant and redundant disk arrays. It does neither model the CPU, host disk controllers nor I/O busses, but only disks [103].

RAMA

RAMA is a ***parallel file system*** that is intended primarily as a cache or storage area for data stored on tertiary storage. Furthermore, RAMA uses hashing algorithms to store and retrieve blocks of a file [54]. See also ***SPIFFI***.

range declustering see ***ViPIOS***

RAPID (Read Ahead for Parallel-Independent Disks)

abstract:

RAPID is a fully ***parallel file system*** testbed that allows implementations of various buffering and ***prefetching*** techniques to be evaluated [90]. The architectural model is a medium to large scale ***MIMD*** **shared memory** multiprocessor with memory distributed among processor nodes. The results represented in [90] show that ***prefetching*** often reduces the total execution time. As a matter of fact, the hit ratio is only a rough indicator of overall performance of a ***caching*** system since it tends to be optimistic and ignores ***prefetching*** overhead [90].

aims:

buffering, ***prefetching***

implementation platform:

RAPID runs on the **Butterfly Plus** multiprocessors (nonuniform memory access time (NUMA) architecture)

data access strategies:

prefetching, *caching*

related work:

disk-directed I/O

people:

David Kotz, Carla Schatter Ellis

{dfk, carla}cs.dartmouth.edu

institution:

Department of Computer Science, Duke University, Durham, NC 27706, USA

http://www.cs.duke.edu/ carla/pario.html

key words:

file system testbed, operating system, disk ***caching***, file system, ***MIMD***

re-redistribution see ***array distribution***

reactive disk and file buffer management see ***SPFS***

read ahead

Communication between clients and servers (or in distributed systems in general) is one of the main overheads in a file system [27]. Hence, I/O requests are packaged, and level locks and resources are managed in groups. Read ahead reads new blocks in advance when a minimum threshold is reached. **Flush ahead** is the opposite of read ahead and frees clean blocks in order to satisfy write requests as soon as possible.

- read ahead: While waiting for client requests, blocks from the servers to the local caches are prefetched in a client. A possible delay time of answers to user request has to be avoided. Hence, ***prefetching*** is executed asynchronously to the user requests. The amount of prefetched data is computed after each read request, and if the number is lower than a determinate value, a thread is used for ***prefetching***. An **Infinite Block Lookahead (IBL)** predictor is used to compute the number of blocks to be read in advance.

- **Write Before Full** (**flush ahead**) is the opposite of read ahead and flushes dirty blocks to the I/O devices before free blocks may be required in the cache. After the execution of a request, the number of dirty blocks is calculated. Only if this number is greater than a fixed threshold, a flush operation is executed asynchronously. The write performance can be increased in this way, but the preflush size has to be considered carefully, because it could cause a delay of the next user request.

redistribution see ***array distribution***

Redundant Array of Independent Disks see ***RAID***

Reed-Solomon codes see ***RAID***

registered objects see ***SHORE***

Remote Memory Access (RMA) see ***MPI-2***

Remote Memory Servers

The memory server model extends the memory hierarchy of multicomputers by introducing a remote memory layer whose latency lies somewhere between local memory and disk [10]. A memory server is a multicomputer node whose memory is used for fast backing storage and logically lies between the local physical memory and fast stable storage such as disks.

Remote Procedure Call (RPC) see ***Agent Tcl, CCFS, SHORE***

request thread see ***SPIFFI***

RMA (Remote Memory Access) see ***MPI-2***

roll-away error recovery see ***SPFS***

ROMIO

abstract:

ROMIO is a high-performance, portable implementation of ***MPI-IO***. A key feature component is an internal abstract I/O device layer called ***ADIO***. The features of ROMIO are described in Chaper 9 of [104], but ROMIO does not support yet: ***shared file pointer*** functions, split collective data access routines, support for file interoperability, I/O error handling, and I/O error classes [145].

aims:

see ***MPI-IO***, ***MPI-2***

data access strategies:

see ***MPI-IO***, ***MPI-2***

implementation platform:

IBM SP, ***Intel Paragon***, HP/Convex Exemplar, **SGI** Origin 2000, Power Challenge and networks of workstations (Sun4, Solaris, IBM, DEC, **SGI**, HP, FreeBSD and Linux)

portability:

PIOFS, ***PFS***, NFS and UNIX file systems (**UFS**)

related work:

ADIO, ***MPI-IO***, ***MPI-2***

people:

Rajeev Thakur, Ewing Lusk, William Gropp

`{thakur, lusk, gropp, romio-maint}@mcs.anl.gov`

institution:

Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, IL 60439, USA

`http://www.mcs.anl.gov/home/thakur/romio/`

key words:

see ***MPI-IO***, ***MPI-2***

round-robin see ***PIOUS, ParFiSys (Parallel File System), SPIFFI, Vesta***

round-robin declustering see ***ViPIOS***

RPC (Remote Procedure Call) see ***Agent Tcl***

S

S-2PL see ***Strict Two-Phase Locking***

scalability guidelines

The ***HFS*** group presents some guidelines in [95] and [93]:

- Preserving parallelism: A demand driven system must preserve the parallelism afforded by the applications. A potential parallelism stems from the application, hence, independent requests should be executed in parallel.
- Bounded overhead: The overhead for each independent service request must be bounded by constants. System-wide ordered queues cannot be used and objects must not be located by linear searches if the queue lengths or search time increase with the size of the system. As a result, this principle restricts growth to be no more than linear.
- Preserving locality: A demand driven system must preserve the locality of the application. Locality is important to reduce the average access time and can be increased by:
    1. properly choosing and placing data structures
    2. directing requests from the application to nearby service points
    3. enacting policies that increase locality in the application's disk access and system requests

Scalable I/O Facility (SIOF)

abstract:

SIOF is a project to enable I/O performance to scale with the computing performance of parallel computing systems and achieve terascale computing [134]. There are three technologies to satisfy the requirements for scalable I/O:

- portable and parallel Application Programming Interfaces (***API***) - ***MPI-IO***
- cost effective ***Network Attached Peripherals (NAPs)***
- Network technology independence

HPSS (High-Performance Storage Systems) is an ***API*** that allows users to set up transfers between multiple storage devices and multiple **compute node**s in a **shared memory** environment.

aims:

demonstration of a network-centered, scalable storage system that supports parallel I/O across the computing environment [134]

implementation platform:

Meiko CS-2 parallel processors (computing nodes), ***IBM SP2*** crosspoint-switched FC fabric to connect the computing nodes

data access strategies:

data is stored and retrieved through the storage system by means of an ***MPI-IO API***

portability:

SIOF is implementing a separate ***API*** on top of **HPSS** for **message passing** architectures and follows the ***MPI-IO*** standard.

institution:

Lawrence Livermore National Laboratory (LLNL), University of California, USA

`http://www.llnl.gov`

key words:

message passing, scalable I/O, ***API***

Scalable I/O Initiative see ***ChemIO, ANL***

scalability see ***CCFS, HiDIOS, Pablo, PIOFS, PVM, scalability guidelines, Vesta, ViPIOS***

Scotch Parallel Storage System (SPFS)

abstract:

Parallel storage systems are constructed as testbeds for the development of advanced parallel storage subsystems and file systems for parallel storage. Scotch has been developing a portable, extensible framework, **RAIDframe**, applicable to simulation and implementation of novel ***RAID*** design [59] in order to advance parallel storage subsystems. The key features are the separation of mapping, operation semantics, ***concurrency control*** and error handling. The file system research is based on ***prefetching*** and ***caching*** techniques. **Transparent Informed Prefetching (TIP)** and the Scotch Parallel File System (SPFS) are the results of the work. The benefit of **TIP** is its ability to increase the I/O concurrency of a single-threaded application [117].

aims:

separated, mechanized, simple, and robust error-handling that does not degrade the performance of error free operations [59] (automated error recovery)

implementation platform:

(experimental testbed)

- Scotch-1 (used for the ***prefetching*** file systems): 25 MHz Decstations 5000/200 with a turbo channel system bus (100MB/s) running the Mach 3.0 operating system; equipped with two SCSI buses and four 300 MB IBM 0661 "Lightning" drives [59]
- Scotch-2 (larger and faster version of Scotch-1; used for the ***RAID*** architecture and implementation and for second generation ***prefetching*** file system experiments): 150 MHz **DEC 3000/500** (Alpha) workstations running the ***OSF/1*** operating system equipped with six fast SCSI bus controllers; each bus has five HP 2247 drives with a total capacity of 30 GB
- Scotch-3 (used for parallel applications, parallel programming tools and multicomputer operation system experiments): 30 **DEC 3000** (Alpha) workstations and 8 **IBM RS6000** workstations. Scotch-3 serves as a storage component in a heterogeneous multicomputer.

data access strategies:

- SPFS supports concurrent read-write sharing within a parallel application
- does not provide synchronization primitives such as barriers or locks
- SPFS anticipates file-sharing and implements a form of weakly consistent **shared memory** [59] by exporting the two primitives "propagate" and "expunge". They are analogous to acquire and release in entry consistency, but lack the synchronization semantics [59].

portability:

SPFS is supposed to complement programming tools such as ***PVM*** or DSM.

related work:

RAID

"advise" system calls in Sun Microsystems' operating system

object-oriented file system called ***ELFS***

disk-directed I/O

application:

[118] discusses experiments including text visualization, database join, speech recognition, object linking and computational physics. The tests were executed on Scotch-3.

people:

Garth A. Gibson, Daniel Stodolsky, Fay W. Chang, William V. Courtright II, Chris G Demetriou, Eka Ginting, Mark Holland,Qingming Ma, LeAnn Neal, R. Hugo Patterson, Jiawen Su, Rachard Youssef, Jim Zelenka, Daniel Stodolsky, M Satyanarayanan, David F. Nagle, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Erik Riedel, Daiv Rochenberg

{garth, danners}cs.cmu.edu

institution:

Parallel Data Lab at School of Computer Science and Department of Electrical Engineering, Carnegie Mellon University, Pitsburgh, Pensilvania, USA

http://www.pdl.cs.cmu.edu/

key words:

disclosed hints, ***prefetching***, ***caching***, file system, I/O management, **shared memory**, distributed file system, ***RAID***

example:

[59] presents an example of an SPFS program where all processes read arbitrary sections of a file and each process writes to a private section of the file.

```
spfs_file_handle sfh;
int my_start = 2000 * process_number();

loop forever
  spfs_read (sfh, ...);
    computation
  spfs_read (sfh, ...);
    computation
    ...
  BARRIER; /*write a disjoint section*/
  spfs_write (sfh, ...);
    computation
  spfs_write (sfh, ...);
    computation
    ...
```

```
      spfs_propagate (sfh, my_start, 2000);
      BARRIER;
      spfs_expunge (sfh, entire_file);
    endloop
```

details:

1 RAID

[42] proposes a graphical programming abstraction for use in standard ***RAID*** development. **Roll-away error recovery** is used in order to eliminate the need for architecture-specific error recovery code. In particular, ***RAID*** operations are represented as **Directed Acyclic Graphs (DAC)**. **RAIDframe** is supposed to allow new array architectures to be implemented.

[42] deals with a fast, on-line recovery failure technique in ***RAID*** level 5. This is especially important for huge on-line applications such as an airline reservation system. The performance-effect of this algorithm comes from a more efficient utilization of the array's excess disk bandwidth. In ***RAID***, **parity logging** is explained.

[155] introduces some ideas concerning ***RAID*** for mobile computers. The driving force for this is the limited amount of electrical power in mobile computers, i.e. they highly depend on battery supply. A power-optimized ***caching*** is a means to overcome the power problem.

2 TIP

Since the traditional **reactive disk and file buffer management** does not meet the needs sufficiently, a **proactive disk and buffer management** is proposed which is based on application-disclosed hints called **Transparent Informed Prefetching (TIP)**. **TIP** offers commands like `tipio_seq` for sequential reading or `tipio_seg` [117]. There are three driving factors for a proactive approach [118]:

- underutilization of storage parallelism
- growing importance of file-access performance

- ability of I/O-intensive applications to offer hints about their future I/O demands

TIP exposes the concurrency in the I/O workload [119] and is able to:

- service multiple I/O requests concurrently
- overlap I/O with computation or "user think time"
- optimize I/O accesses over a large number of outstanding requests

The predictability of an application's access pattern could be used to inform the file system of future demands on it rather than initiating asynchronous I/O. 'Disclosure' is in contrast to 'advice' where a programmer's knowledge is exploited to recommend how resources should be managed [118]. Disclosing hints are issued through an I/O-control (ioctl) system call.

[116] outlines the difference between disclosure and advice by some examples:

Hints that disclose:	"I will read file F sequentially with stride S"
	"I will read these 50 files serially and sequentially"
Hints that advise:	"cache file F"
	"reserve B buffers and do not read-ahead"

The main reasons for not applying advice are that users are not qualified to give advice, and an advice is not portable.

TIP2 uses a cost-benefit-analysis to allocate global resources among multiple processes [146]. In particular, decisions can be made by weighing the benefits of providing resources to a costumer against the cost of taking them from a supplier [146]. An algorithm called **TIPTOE** quantifies the costs and benefits of deeper ***prefetching***.

3 SPFS

SPFS is based on a **client-server** model where the client processes interface directly with servers through a portable library. SPFS also uses **TIP** for aggressive ***data prefetching***. The level of fault-protection can also be chosen by the application due to a redundancy on a per-file basis, i.e. redundancy computation can be dis- and enabled to minimize the performance cost of short bursts of rapid changes [59].

4 Network-attached Storage

[57] describes the usage of a network-attached storage which provides high performance by directly attaching storage to the network, avoiding **file server** store-and-forward operations and allowing data transfer to be stripped over storage and switched-network links. In brief, a command interface reduces the number of client-storage interactions that must be relayed through the file manager, offloading more of the file manager's work without integrating file system policy into the disk [57].

scripting language see ***Agent Tcl***

SDL (SHORE Data Language) see ***SHORE***

SDDF (Pablo Self-Describing Data Format) see ***Pablo***

sequential I/O interface see ***I/O interfaces***

serializability see ***PIOUS, PPFS***

server-directed I/O see ***Panda (Parallel AND Arrays)***

SGFP (synchronous-global file pointer) see ***SPIFFI***

SGI see ***ChemIO, HFS, ROMIO***

shadowing see ***RAID***

Shared File Model see ***CHANNEL***

shared file pointer

A shared file pointer is much more powerful than a traditional private or local UNIX file pointer since it can simplify the coding and increase the performance of parallel applications [54]. As for a shared file pointer, it is ensured that a file is read sequentially even if many processes share the same file. Additionally, it reduces the number of disk seeks and increases the effectiveness of ***prefetching*** [54].

shared memory see ***Cray C90, Fujitsu AP1000, Global Arrays, HFS, I/O in-***

terfaces, Intel Paragon, Kenal Square, MIMD, Multipol, p4, PVM, RAPID, Scalable I/O Facility, Scotch Paralle File System, Shared Virtual Memory

shared nothing see ***SHORE***

Shared Virtual Memory

Shared Virtual Memory implements coherent **shared memory** on a multicomputer without physically **shared memory** [10]. The **shared memory** system presents all processors with a large coherent **shared memory** address space. Any processor can access any memory location at any time. See also ***GPM***.

SHORE (Scalable Heterogeneous Object REpository)

abstract:

SHORE is a persistent object system (under development) that represents a merger of ***object-oriented database (OODB)*** and file system technologies [21]. The work is based on ***EXODUS***, an earlier object-oriented data base effort, but it differs in the following points [21]:

- A storage object contains a pointer to a type object that defines its structure and interface.
- architecture: SHORE has a symmetric **peer-to-peer** structure, and it supports the notion of a value added [21] server.
- SHORE is much more a complete system than an **EXODUS storage manager (ESM)**.

The **ParSet** facility of SHORE provides a means of adding ***data parallel*** executions to **OODBMS** applications [48].

aims:

provide a system that addresses ***OODB*** issues and enabling "holdout" applications to finally move their data out of files into persistent storage objects

implementation platform:

shared-nothing multiprocessor

portability:

PVM is used for interprocess communication

related work:

EXODUS

application:

Pointer swizzling can improve the performance of **OODBMS**s while accessing persistent objects that have been cached in main memory [150]. In particular, the performance increase stems from converting pointers from their disk format to an in-memory format when objects are faulted into memory by the **OODBMS**. SHORE was also used to design ***Paradise***, a database system for handling GIS (Geographical Information System) applications.

people:

Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, Set J. White, Michael J. Zwilling

`{shore, paradise}@cs.wisc.edu`

institution:

Computer Science Department, University of Wisconsin, Madison, WI 53706, USA

`http://www.cs.wisc.edu/shore/`

key words:

persistent object system, **shared-nothing**, object-orientated DB, file system

example:

[21] demonstrates how an **SDL** file (see below) is transferred into a C++ class.

oo7.SDL:

```
Module oo7{
const long TypeSize = 10;
enum BechmarkOP {Trav1, Trav2, Trav3, etc};

//forward declarations
interface Connection;
interface CompositePart;

interface AtomicPart {
```

```
public:
  attribute char ptype[TypeSize];
  attribute long x,y;
  relationship set<Connection> to inverse from;
  relationship set<Connection> from inverse to;
  relationship ref<CompositePart> partOf inverse parts;
  void swapXY();
  long traverse (in BechmarkOp op, inout PartIdSet visitedIds) const;
  void init( in long ptId, in ref<CompositePart> cp);};
}
```

C++ class generated from oo7.SDL

```
class AtomicPart {
public:
  char ptype[109];
  long x, y;
  Set<Connection> to;
  Set<Connection> from;
  Ref<CompositePart> partOf;
  virtual long traverse (BenchmarkOp op, PartIdSet &visitedIds) const;
  virtual void init (long ptId, Ref<CompositePart> cp);
};
```

Information about an atomic part could be printed by a C++ function as follows:

```
void printPart (Ref<AtomicPart> p {
  cout << "Type" << p->ptype << part at (" << p->x ",
                              "<< p->y << )\n";
}
```

details:

1 Basic Concepts

SHORE is a collection of cooperating data servers, and a UNIX-like namespace is provided. In contrast to UNIX, each object can be accessed by a globally unique Object Identifier (OID). What is more, there are also a few new features of objects like types and pools.

Like in a database object model, the SHORE object model consists of objects and values. In order to obtain flexibility of dynamic data structures, SHORE allows objects to be extended with a variable-sized heap. The two main services of the file system are object naming and space management. The namespace also allows the usage of **anonymous objects**, which do not have path names, but can be accessed by OID like any other objects. UNIX-like objects are called **registered objects**.

SHORE types are defined in the **SHORE Data Language (SDL)**. Moreover, a database built in one language should be accessed and manipulated by applications in other OO languages. An application can be created as follows [21] - C++ bindings are operational:

- write a description of the types in **SDL**
- use the **SDL** compiler to create type objects corresponding to the new types
- use a language specific tool to derive a set of class declarations and special-purpose function definitions from the type objects

Other services provided by SHORE are ***concurrency control***, crash recovery, optimized object queries and a flexible, user controllable notion of sticky objects to permit users to cluster related objects.

2 Architecture

SHORE executes as a group of communicating processes called SHORE servers [21]. Each of these servers has several capabilities [21]:

- It is a page-cache manager.
- It acts as an **agent** for local **application process**es. In particular, an **RPC** call is sent requesting to the local server, which fetches the necessary page(s) and returns the object.
- It is responsible for ***concurrency control*** and recovery.

The basic parallel construct is **ParSet (Parallel Set)** which can adopt the ***data parallel*** approach to object-oriented programming. **ParSet** is a set of objects of the same type and uses **SDL** as the type language for **ParSet** objects. Additionally, primary and secondary **ParSet**s can be distinguished. The **master-slave** model is expanded, and one node runs on an additional **ParSet Server (PSS)** [48].

SHORE Data Language (SDL) see ***SHORE***

SIMD (Single Program Multiple Data)

SPMD is a model for large-scale scientific and engineering applications. The same program is executed an each processor, but the input data to each of the programs may be different [16].

The most widely used classification is the one where the von Neumann model is viewed as a Single Stream of Instructions controlling a Single Stream of Data (**SISD**). One instruction produces one result and, hence, there is a Single Instruction Stream and a Single Data Stream. One step towards parallelism leads to the SIMD model, another step ends up with Multiple Instruction Streams (***MIMD***). In the classical example of a parallel SIMD model, a number of identical **processing element**s receive the same instruction broadcast by a higher instance. Each **processing element** performs the instruction on its own data item. In other words, a SIMD instruction means that a Single Instruction causes the execution of identical operations on Multiple pairs of Data [3]. Furthermore, this is the simplest conceptual model for a vector computer. The synchronization can be obtained by using a broadcast command that keeps the processes in a lockstep, and the processes need to talk to each other for synchronization purpose. Additionally, they need not store their own programs, which results in a smaller design and a bigger amount of processes.

A SIMD machine contains many data processors operating synchronously, each executing the same instruction and using a common program counter [81]. Furthermore, each processor (**processing element, PE**) is a fully functional ALU (Arithmetic Logical Unit). Many **PE**s are called **PE** array, which can be treated as a linear array or as an array of higher dimensions. Moreover, each **PE** has an execution flag indicating whether the **PE** should execute the current instruction. A **front end (FE) processor** has to drive the **PE** array by broadcasting instructions and related data to all **PE**s. Additionally, it has to perform all scalar computations and control flow operations, and the **FE** is the system interface to the external environment. What is more, all **PE**s are connected by an interprocessor communication network, which allows the **PE**s to access data stored within the memories of other **PE**s.

SIMD machines offer impressive cost/performance ratios, and are well suited for a large body of engineering and scientific applications [81].

simple-strided access see ***disk-directed I/O***
SIOF see ***Scalable I/O Facility***
SISD see ***SIMD***
slab see ***PASSION, data sieving, data prefetching***
small request see ***RAID-I***
SMP Digital Alpha see ***CHAOS***
space preallocation see ***ParFiSys (Parallel File System)***
SPARC see ***ChemIO, Fujitsu AP1000, OPT++, TPIE***
SPFS see ***Scotch Parallel File System***

SPIFFI (Scalable Parallel File System)
abstract:

SPIFFI is a high-performance ***parallel file system*** that stripes files across multiple disks.

aims:

intended for the use of extremely I/O intensive applications

implementation platform:

SPIFFI is the ***parallel file system*** for the ***Intel Paragon***

data access strategies:

SPIFFI provides applications with a high-level flexible interface including one individual and three ***shared file pointer***s.

portability:

A library of C functions can be used to access SPIFFI files.

related work:

RAID, ***RAID-II***, ***disk-directed I/O***, ***PFS***, ***Vesta***, ***CMMD I/O system***, ***RAMA***, ***Bridge parallel file system***

people:

Craig S. Freedman, Josef Burger, David J. DeWitt

`{freedom, bolo, dewitt}@cs.wisc.edu`

institution:

Computer Science Department, University of Wisconsin, Madison, WI 53706, USA

`http://www.cs.wisc.edu/shore/`

key words:

file system, **distributed memory**

details:

A file is partitioned across disks horizontally in a **round-robin** fashion. When a user creates a file, the a set of disks and the stripping granularity have to be stated. The portion of a file stored at one disk is called file fragment. Moreover, each disk node has a file system which is responsible for mapping local file blocks to physical file blocks and for recording each fragment's size. In addition, each node also caches a copy of the **meta-data** for each file to open.

The **local file pointer (LFP)** is a single process and allows reading or writing an entire file sequentially or accessing random portions of it [54]. It is comparable to ***PFS***'s `M_UNIX` I/O mode. The three ***shared file pointer***s are called **global file pointer (GFP)**, **synchronized-global file pointer (SGFP)** and **distributed file pointer (DFP)**. The **GFP** is shared among a group of processes which access the same file. It enables a collective read or write. The file pointer can be compared to ***PFS***'s `M_LOG` I/O mode. Processes that share a **SGFP** (comparable to `M_SYNC` in ***PFS***) are assigned a fixed cyclical ordering and may only access the file pointer in that order. **DFP** is intended for I/O intensive applications.

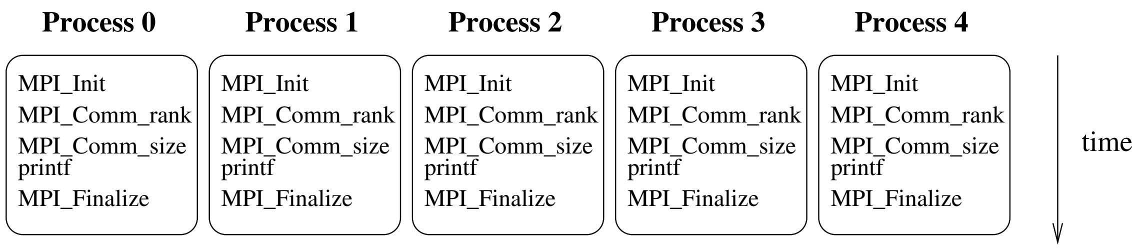


Figure 2.25: SPMD programming model

SPIFFI also employs three types of threads. Each disk runs a **request thread**, a **control thread** and a **GFP thread**. The **request thread** is responsible for receiving read and write requests, the **control thread** manages file operand close requests, and the **GFP thread** provides atomic read and write update operations. What is more, each buffer pool is allocated at each disk node to improve application performance [54].

split-phase interface see ***Multipol***

SPMD (Single Program Multiple Data)

In ***MPI***, only one program is written which is executed on each of the processes that compete in the calculation. In the example (see ***MPI***), five copies of the program are run at the same time on five different processes (see Figure 2.25).

Process 0 executes its program as well as processes 1, 2, 3 and 4. Each process has its own program and works independently of the other ones. It does not even have to know that there are other processes as well that execute the same program. A process executes its program in a sequential way and produces an output which indicates its process id. On the screen, the final result of five calculations can be seen. As in a real parallel computation model, the order in which the output appears is not defined. ***MPI*** does not specify how such a parallel computation is started and, therefore, such programs are non-deterministic. For instance, process 4 could execute the print instruction before process 2 or vice versa. The correct output cannot be determined before the execution of the program [135].

STARFISH

STARFISH is a parallel file system simulator (developed at Dartmouth College, USA) which ran on top of the Proteus parallel architecture simulator, which in turn ran on a **DEC-5000** workstation [88]. See also ***disk-directed I/O***.

storage node see ***Vesta***

storage object see ***Hurricane File System (HFS)***

storage server see ***RAID-II***

Strict Two-Phase Locking (S-2PL)

abstract:

> The most common form of a scheduler is based on strict two-phase locking (S-2PL) [110]. Under S-2PL, before an **I/O daemon** can perform a data access, a read or write **lock** must be obtained. If the data is currently locked and the **lock** types will **conflict**, then the data access must be delayed until the **lock** can be obtained. Two locks on the same data item **conflict** if issued by different transactions and one or both is a write **lock**. S-2PL results in a serializable schedule.

details:

S-2PL is also well known in the data base world. There a transaction is a finite sequence of data access operations that translates the database from one consistent state to another one. Transactions have to be executed in a form so that the effect is as if transactions are performed in a serial or sequential order. Transactions that satisfy these conditions are said to be serializable.

When an application executes a read or a write library function call, a unique transaction identifier is generated for tagging all messages to be sent. It is then determined which **I/O node**s contain the file data to be accessed. Messages are sent to the appropriate **I/O daemon**s requesting that data to be read or written. Independently, each **I/O daemon** satisfies each request, after obtaining the necessary **lock**, and replies with the result. Write requests must be performed so that the initial values can be stored if the transactions fail (***check-***

pointing). All results are collected and, if they all indicate success, a **commit** message is then sent to each participating **I/O daemon**; **abort** otherwise. Thereafter, control is returned to the caller. Having received a **commit** message, an **I/O daemon** makes permanent any write access and frees all locks held on its behalf.

The disadvantage of S-2PL is that **deadlock**s are possible when a **circular wait** condition develops. A simple way to solve this problem is that each daemon has to time-out a request that has been delayed too long on a **lock** [110]. See also ***PIOUS***.

strided access see ***disk-directed I/O, Portable Parallel File System (PPFS)***
strided segment see ***disk-directed I/O***
stripe unit see ***PPFS, RAID, RAID-I***
stripmining see ***PASSION***
striping unit see ***design of parallel I/O software, RAID***
Sun 4/280 see ***RAID-I***

supercomputing applications

Supercomputing applications are generating more and more data, but I/O systems cannot keep abreast, i.e. they become less able to cope with the amount of information in a sensible amount of time. The solution requires correct matching of bandwidth capability to application bandwidth requirements, and using of buffering to reduce the peak bandwidth that I/O systems have to handle.

Conventional file systems use ***caching*** for reducing I/O bandwidth requirements. Thus, the number of requests can be decreased, and the system performance is increased. Another method of reducing I/O is the usage of **delayed write**s. A **write-behind cache policy** is required, which allows a program to continue executing after writing data to the cache without waiting for the data to be written to the disk.

The environment of a supercomputer (e.g. **Cray Y-MP** 8/832) is different from a conventional one. It is characterized by a few large processes that consume huge amounts of memory and CPU time. Jobs are not interactive, but submitted in batch and run whenever the scheduler can find enough resources.

Supercomputers are ideal for applications that require the manipulation of large arrays of data. They are especially applied in fields like fluid dynamics, structural dynamics or seismology [106].

[18] presents six types of I/O that can be identified with typical computational science applications: input, output, accessing **out-of-core** structures, debugging, scratch files and checkpoint/restart.

Synchronized Access see ***Portable Parallel File System (PPFS)***
synchronous-global file pointer (SGFP) see ***SPIFFI***

T

T800 see ***ParFiSys (Parallel File System)***
T9000 see ***ParFiSys (Parallel File System)***

task parallel program

A task parallel program consists of a set of (potentially dissimilar) parallel tasks that perform explicit communication and synchronization [4]. **Fortran M (FM)** and **CC++** are examples of such a language.

Tcl see ***Agent Tcl, TIAS***
TCP/IP see ***TIAS***

TIAS (Transportable Intelligent Agent System)

abstract:

Transportable **agent**s fall in the intersection between the fields of intelligent **agent**s and the field of remote computation [65]. What is more, it can transport itself as well as communicate with other **agent**s. The system is layered to enhance modularity.

aims:

find abstract methods of manipulating on-line data that serve the needs of end users efficently and use network resoucres intelligently [65]

implementation platform:

DEC MIPS / Ultrix **DEC Alpha** / ***OSF/1***

related work:

Agent Tcl

portability:

relatively portable to UNIX platforms

application:

- calendar manager: A mobile-**agent** script travels from calendar to calendar, requesting availability information and attempts to secure a reservation for a certain time.
- daily newspaper
- monitoring: stock trading **agent**s can handle a portfolio; monitoring WWW documents for changes

people:

Kenneth E. Harker

`iago@cs.drtmouth.edu`

institution:

Department of Computer Science, Dartmouth College, Hanover, HN 03755-3510, USA

key words:

transportable **agent**

details:

The TIAS implementation can be divided into three layers of code:

1. The transportation layer is at the base and is responsible for moving an **agent**. **TCP/IP** and sockets are used to transport an **agent** from one machine to another.

2. The interpreter layer interprets the **agent** script which is written in C. This allows the programmer to add special routines and services which **Tcl** cannot perform.

3. The script layer is written in **Tcl** and is the actual 'agent'.

Agents are separated into three distinct types: host, resource-managing and mobile **agent**s. The host **agent** should receive 'register' and 'unregister' messages from mobile **agent**s and record the identifiers of the **agent**s on the system at that point in time. Resource-managing **agent**s have access to certain information that they might choose to share with other **agent**s or might choose to allow other **agent**s to modify.

TIP (Transparent Informed Prefetching) see ***SPFS***

TIP2 see ***SPFS***

TIPTOE see ***SPFS***

TOPs (The Tower of Pizzas)

abstract:

TOPs is a portable software system providing fast parallel I/O and buffering services [140].

aims:

provide parallel access to data striped across nodes/workstations

exploit ***caching*** and ***prefetching*** to diminish latency

be efficient and portable

implementation platform:

IBM SP2

related work:

Zebra (**client-server** architecture, using a log structured file system)

SPIFFI implements global ***shared file pointer***s of varying flavors

Jovian, ***ADOPT***, ***RAID-II***

people:

Michael Tan, Nick Roussopoulos, Steve Kelley

{mdtanx, nick, skelly}cs.umd.edu

institution:

Institute for Advanced Computer Studies (UMIACS), Computer Science Department, University of Maryland, College Park, MD 20742, USA

key words:

parallel software

details:

The system architecture is a collection of nodes connected by a fast network. Furthermore, each of these nodes has a CPU, large memory and one or more large disks. A collection of workstations on a fast network as well as a multiprocessor machine can fulfill the needs. A **peer-to-peer** architecture similar to a distributed file system is used [140]. A TOPs process runs on each node and is responsible for servicing requests, striping over the network, and managing several local resources. What is more, the system can be logically partitioned into a **client-server** architecture.

TPIE (Transparent Parallel I/O Environment)

abstract:

TPIE is designed to allow programmers to write high performance I/O-efficient programs for a variety of platforms [149]. The work on TPIE is still in progress. Moreover, TPIE has three main components: **Access Method Interface (AMI), Block Transfer Engine (BTE)** and **Memory Manager (MM)**.

aims:

support the implementation of high performace I/O-efficient programs

implementation platform:

The following combinations have been tested:

- Sun **SPARC**station - SunOS 4.x

- Sun **SPARC**station - Solaris 5.x
- **DEC Alpha** - ***OSF/1*** 1.x and 2.x
- HP 9000 - HP-UX
- Intel Pentium - Linux 1.x

people:

Darren Eric Vengroff

institution:

three institutions take part in the TPIE project:

Department of Computer Science, Duke University, USA

Department of Electrical Engineering, University of Delware, USA

Center for Geometric Computation, Duke University, USA

key words:

I/O environment

details:

The three components can be seen as three different layers (see Figure 2.26). The **AMI** implements fundamental access methods (scanning, ***permutation*** routing, merging, sorting, ***distribution***, and batch filtering). Furthermore, it provides an OO interface to application programs. The **MM** has to manage main memory, including memory distributed across multiple machines, i.e. it manages random access memory on behalf of TPIE [149]. The **BTE** is the interface between the I/O hardware and the rest of the system. In particular, it is responsible for moving blocks of data from physical disks to main memory and back.

TPIE programs work with streams of data stored on disks [149]. The creation of a stream of objects is similar to creating an object in C++. A simple example illustrates the creation of an integer stream:

```
AMI_STREAM<int> my_stream;
AMI_STREAM<int> *my_stream = new AMI_STREAM<int>;
```

The program fragment also demonstrates the usage of the **AMI**.

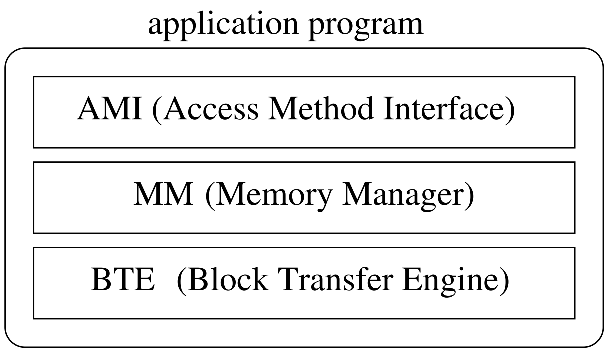


Figure 2.26: TPIE

translation table see ***PARTI***

Transputer see ***ParFiSys (Parallel File System)***

tuple space see ***Local File Server***

Two-Phase Method (TPM)

abstract:

PASSION introduces a Two-Phase Method which consists of the following two phases:

- READ DATA (processes cooperate to read data in large chunks)
- DISTRIBUTE DATA (interprocess communication is used so that each processor gets the data it requested)

details:

1 TPM for In-core Arrays

I/O performance is better when processors make a small number of high granularity requests, instead of a large number of low granularity requests. ***PASSION*** performs **collective-I/O** using this TPM [33]. The advantage is that it results in high granularity data transfer between processors and disk, and it makes higher bandwidth of the processor interconnection network.

2 Extended TPM (ETPM) for OOC arrays

It can be used for accessing data in both ***Global Placement Model*** and ***Partitioned In-core Model***. This method performs I/O for **OOC** arrays efficiently by combining several

I/O requests into fewer larger requests, eliminating multiple disk accesses for the same data, and allows the access of arbitrary sections of **out-of-core** arrays [143].

The **collective-I/O** interface says that all processors must take part in the communication process (must call the **ETPM** read/write routine) even if they do not need data (request for 0 bytes). The advantage of this concept is that all processors can cooperate to perform certain optimizations since all processors are participating.

ETPM assigns ownership to portions of the file called **File Domain (FD)**, and a processor directly accesses only the portions of the file it owns.

The two phases are:

1. All processors exchange their own access information with all other processors so that each processor knows the access requests of all other nodes. This information is stored in a data structure called **File Access Descriptor (FAD)** and contains exactly the same information on all nodes. A **File Domain Access Table (FDAT)** of all processors contains information about which section of its **FD** have been requested by other processors.

2. Communicating the data that has been read in the first phase to the respective processors. The information in the **FDAT** is sufficient for each processor to know what data to be sent to which processor.

The whole **ETPM** algorithm looks like follows [143]:

1. Exchange access information with other processors and fill in the **FAD**.

2. Compute intersections of **FD** and this processor's **FD** and fill in the **FDAT**.

3. Calculate the minimum of the lower-bounds and the maximum of the upper-bounds of all sections in the **FDAT** to determine the smallest section containing all the data needed from the **FD**.

4. Read this section using ***data sieving***.

5. Communicate the data to the requesting processors.

Advantage: If, for example, each processor needs to read exactly the same section of the array, it will be read only once from the file and then broadcast to other processors over the interconnecting network. The algorithm for writing sections in **OOC** arrays is essentially the reverse of the algorithms described above.

type map see ***MPI***

U

UFS (UNIX File System) see ***PFS, ROMIO***
UnCVL see ***CVL***

UNIX I/O

The UNIX I/O facility can be applied in a uniform way to a large variety of I/O services, including disk files, terminals, pipes, networking interfaces and other low-level devices. Many application programs use higher-level facilities, because they have more specified features. An example is the standard I/O library `stdio` for C, which is an application-level facility (see ***I/O interfaces***). The functions correspond to the UNIX functions.

V

Vesta

abstract:

Vesta is a ***parallel file system*** providing parallel file access to application programs running on multicomputers with parallel I/O subsystems [36]. A file can be divided into partitions (multiple disjoint sequences). Furthermore, Vesta allows a direct access from a **compute node** to the **I/O node** without referencing any centralized metadata. Consequently, Vesta is based on a **client-server** model, which allows libraries to be implemented on top of Vesta [37].

aims:

The main goal is to provide high performance for I/O intensive scientific applications on massively parallel multicomputers.

implementation platform:

Vulcan multicomputer at the IBM T. J. Watson Research Center

implementation on an **IBM SP1**

applications on top of **IBM RS/6000**

workstations [8] a **message passing** library MPX (closely related to ***MPL***) is also installed [8]

data access strategies:

switch between ***concurrency control*** and no ***concurrency control***

synchronous/asynchronous read/write

prefetching data in advance

blocking/non-blocking I/O commands (overlapping computation and I/O)

related work:

partitioning of files (see below) is also provided by the ***nCUBE*** file system, but Vesta is supposed to be more flexible [37]; Vesta is the basis for the ***PIOFS*** parallel I/O file system.

application:

four applications are studied in [8]: matrix, sorting, seismic migration and video server applications

people:

Peter F. Corbett, Jean-Pierre Prost, Sandra Johnson Baylor, Tony Bolmarcich, Dror Feitelson

`{corbett, jppost, sandyj}@watson.ibm.com`

institution:

IBM T. J. Research Center, Yorktown Heights, NY 10579, USA

http://www.research.ibm.com/people/c/corbett/vesta.html

key words:

client-server, **distributed memory**, ***MIMD***, file system, views

example:

Vesta uses a model for partitioning of files. This can be easily adapted to matrix multiplication where a whole row or a whole column can be referred to as a partition unit. Obviously, this makes matrix multiplication easier and can be done similar to vector operations. The following example from [37] multiplies two matrices A and B and stores the result in C:

```
pdim = sqrt (PROC_NUM);
blk_size = MAT_SIZE / pdim;
float A[MAT_SIZE][blk_size],
      B[blk_size][MAT_SIZE],
      C[blk_size][blk_size];
Vgs = blk_size; Vi = pdim;
Hgs = MAT_SIZE; Hi = 1;
fda = Vesta_Open ("A", Vgs, Hgs, Hi, me / pdim);
Vgs = MAT_SIZE; Vi = 1;
Hgs = blk_size; Hi = pdim;
fdb = Vesta_Open ("B", Vgs, Vi, Hgs, Hi, me % pdim);
Vgs = blk_size; Vi = pdim;
Hgs = blk_size; Hi = pdim;
fdc = Vesta_Open ("C", Vgs, Vi, Hgs, Hi, me);
Vesta_Read (fda, A, MAT_SIZE * blk_size);
Vesta_Read (fdb, B, MAT_SIZE * blk_size);
for (i = 0; i < blk_size; i++)
{
   for (j = 0; j < blk_size; j++)
   {
     C[i][j] = 0;
     for (k = 0; k < MAT_SIZE; k++)
        C[i][j] += A[k][j] * B[i][k];
   }
}
Vesta_Write (fdc, c, blk_size*blk_size);
```

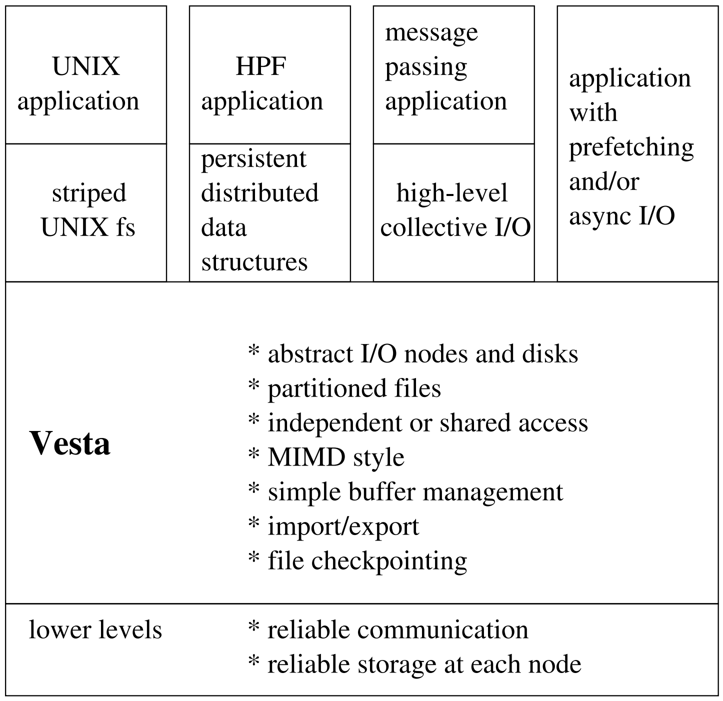
details:

[36] and [37] present some design guidelines for ***parallel file system***s:

- Parallelism: This is most important for high performance and can be achieved by providing a parallel interface to eliminate any points where access is serialized. In particular, multiple **compute node**s access multiple **I/O node**s at the same time and independently from each other.
- **Scalability**
- Layering, Reliability: Vesta is based on a layered model (see Figure 2.27) and establishes the middle layer between applications and disk [36], i.e. high level components have to be implemented on top of Vesta (e.g. Vesta has no support for **collective-I/O** - this has to be implemented by a library on top of Vesta). On the other hand, Vesta is reliable to the underlying lower layers that have to provide services such as **message passing** and error checking/correction (***RAID***). However, it provides additional features like ***checkpointing*** of files and allows files to be exported for safekeeping elsewhere.
- Rich features: Vesta provides services such as a hierarchical structure of directories, permission bits for each file owner and suchlike, but they are not fully compatible with existing systems.
- ***MIMD*** style

Vesta is implemented in two sub-units: On the one hand, there is a client library linked with applications running on the **compute node**s, on the other hand, there is a server running on the **I/O node**s. The file **meta-data** is distributed across all **I/O node**s. In contrast to UNIX, Vesta hashes the file name in order to obtain the **meta-data** rather than consulting directory entries and the corresponding **i-node**s. This **meta-data** is only accessed when the file is attached to the application [36]. Another difference to existing systems is that data is not cached on **compute node**s. Instead, memory buffers are used in the **storage node**s [37].

The parallel structure of a file is defined at two levels by the interface: the normal, physical view (how many **storage node**s are used) and the logical view of a file. Vesta does not stick



UNIX application
HPF application
message passing application
application with prefetching and/or async I/O
striped UNIX fs
persistent distributed data structures
high-level collective I/O
Vesta
* abstract I/O nodes and disks
* partitioned files
* independent or shared access
* MIMD style
* simple buffer management
* import/export
* file checkpointing
lower levels
* reliable communication
* reliable storage at each node

Figure 2.27: Vesta's place in the layered system

to the physical layer of a traditional file system, but uses a 2-D structure [37] where a file is composed of a certain number of physical partitions, each of which is a linear sequence of records. This number can correspond to the number of **storage node**s. If the number exceeds the amount of **storage node**s, the partitions are distributed across the **storage node**s in a **round robin** fashion. A physical partition is also referred to as **cell** (especially in more recent papers on Vesta). In particular, a **cell** can be seen as a container where data can be deposited or as a virtual **I/O node** [36].

The logical view is independent of the physical view and allows the file to be viewed at differently without changing the physical structure. Such a logical view is set when the file is opened. Moreover, a view is based on a 2-D template, and all records falling under a specific template belong to a distinct logical partition of the file [37]. Consequently, the data in a file has no unique sequence. What is more, each **cell** can have a different depth.

In UNIX the system call `open` can be used for opening existing files as well as for creating

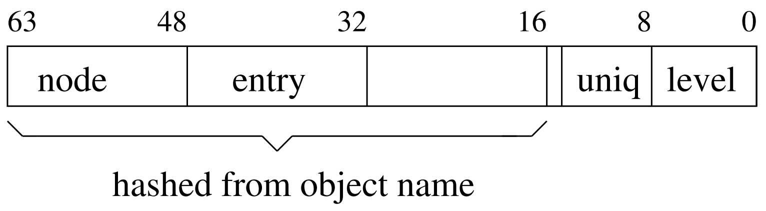


Figure 2.28: Structure of the 64-bit internal ID of Vesta objects

new files. Vesta provides equivalent operations, but not with a single instruction: opening a file is always done together with appending a file. Thus, Vesta provides three separate functions `Vesta_create`, `Vesta_open` and `Vesta_attatch`. Data access functions can be used in a CAUTIONS or a RECKLESS way, which corresponds to a flag that is issued when using file access functions. In a CAUTIONS access, ***concurrency control*** is guaranteed. However, this can lead to performance cost. Hence, Vesta allows to neglect an atomically execution of file access operations, especially in cases where accesses are sure to be non-conflicting [37]. ***PIOUS*** applies a similar concept by using stable and volatile transactions.

Although Vesta has directories, a file name is hashed into a 48-bit value in order to get the **meta-data** of the file. This 48-bit value is important for the 64-bit internal ID. 16 of the 48 bits are hashed to obtain the **I/O node** that serves as a **master node** for the file. The next 16 bits define the object table on the **master node**. To sum up, the 64-bit ID contains a unique ID, the file name, its owner ID, group, and access permissions, creation, access, and last modification times, the number of **cell**s, the number or units within the **cell**, and the current file status [36]. The last 8 bits (called level) number the **cell**s of a file on a given **I/O node** (see Figure 2.28).

The directory mentioned above is only useful for the user to organize the files in sublists, but is not required by the file system for file **meta-data**. However, [36] also adds some drawbacks concerning the use of hash tables:

- no control over access using directory permission bits
- no links

- renaming is a time consuming process

VFCS (Vienna Fortran Compilation System) see ***Vienna Fortran, ViPIOS***

ViC* (Virtual-Memory C*)

abstract:

ViC* initially was a compiler-like preprocessor for **out-of-core** ***C**** but [35] refers to it as a compiler. The input is a ***data parallel C**** program with parallel variables and certain shapes, and the output is a standard ***C**** program [39]. ViC* analyzes program data flow and performs program transformation to reduce I/O demands [39], calculates the amount of in-core data to make full use of the available memory, and fuses in-core sectioning loops to avoid repeated transfers of the same data.

aims:

exploit existing languages and software as much as possible

C* has been chosen, because the ***data parallel*** language is used at several sites [39]

ViC* is expected to be a testbed for parallel I/O research

related work:

HPF, ***Vienna Fortran***, ***Fortran D***

people:

Thomas H. Cormen, Alex Colvin

`thc@cs.dartmouth.edu, alex.colvin@dartmouth.edu`

institution:

Department of Computer Science, Dartmouth College, Hanover, NH, 03755-3510, USA

`http://www.cs.dartmouth.edu/ thc/vic.html`

key words:

out-of-core, precompiler, ***C****, data parallelism, **virtual memory**

example:

[39] presents the following ViC* example which computes a truncated harmonic series and normalizes it.

```
#define N (1L<<40)
```

```
outofcore shape [N]series;
float:series  normal;

void main ()
{
  float:series harmonic;
  float sum;
  int:series k;

  with (series)
  {
     k = pcoor(0);
     where (k>0)
     {
       harmonic = 1.0 / k;
       sum= += harmonic;
       normal = harmonic * (1.0 / sum);
       [.-1]normal = normal;
     }
  }
}
```

Vienna Fortran (VF)

Vienna Fortran is a ***data parallel*** language which supports the ***SPMD*** model of computation. Furthermore, it provides explicit expressions for data mapping [12]. The corresponding compiler is called **Vienna Fortran Compilation Systems (VFCS)**.

Five major steps are required to transform a Vienna Fortran **out-of-core** program into a **out-of-core** ***SPMD*** program [19]:

1. ***distribution*** of each **out-of-core** array among the processors
2. ***distribution*** of the computation among the processors
3. splitting execution sets into tiles
4. insertion of I/O and communication statements
5. generation of a Section Access Graph (SAG)

Vienna Fortran Compilation System (VFCS) see ***Vienna Fortran, ViPIOS***

VIP-FS (VIrtual Parallel File System)

abstract:

VIP-FS is a straight-forward interface to parallel I/O [46]. It is virtual because it is implemented using multiple individual standard file systems integrated by a **message passing** system. VIP-FS makes use of **message passing** libraries to provide a parallel and distributed file system which can execute over multiprocessor machines or heterogeneous network environments.

aims:

Portability is a very important objective in VIP-FS, because it must be portable across different libraries. Hence, features of the most common **message passing** libraries must be employed. What is more, it has to be able to operate in heterogeneous distributed systems.

implementation platform:

Ethernet on the **IBM SP1**

data access strategies:

VIP-FS employs three strategies:

- Direct Access: Every I/O request is translated into requests to the appropriate I/O device.
- Two-Phase Access (see ***TPM***)
- Assumed Requests

portability:

an ***MPI*** compatible interface is being planned

people:

Juan Miguel des Rosario, Michael Harry, Alok Choudhary

`{mrosario, mharry, chaudhar}@npac.syr.edu`

institution:

Northeast Parallel Architecture Center, 111 College Place, RM 3-210, Syracuse University, USA

key words:

file system, data ***distribution***, parallel architecture, **message passing**, ***distributed computing***

example:

The following fragment illustrates code segments [46] for data decomposition and parallel file mapping specification:

```
mapinfo_T info;
fd = pvfs_open ("testfile", O_CREATE|O_WRONLY|O_TRUNC, 0666);

info.globilize[0] = 2048;
info.globalize[1] = 2048;
info.discode[0] = 0;    // row-block
info.distcode[1] = 1; // column is distributed
info.blocksize[0] = 0;
info.blocksize[1] = 0;
info.nprocs[0] = 1; // proces per row
info.nprocs[1] = 4;   // procs per column
// load data distribution
info.status = pvfs_ioctl (fd, IOCTL_LOADADT, &info);

info.globilize[0] = 2048;
info.globalize[1] = 2048;
info.discode[0] = 0;    // row-block
info.distcode[1] = 1; // column is distributed
info.blocksize[0] = 0;
info.blocksize[1] = 0;
info.nprocs[0] = 0; // proces per row
info.nprocs[1] = 1;   // procs per column
// load strip-file distribution
info.status = pvfs_ioctl (fd, IOCTL_LOADFDT, &info);
```

details:

VIP-PF has three functional layers:

- **Interface layer**: provides a variety of file accesses
- **virtual parallel file layer (VPF)**: defines and maintains a unified **global view** of all file system components

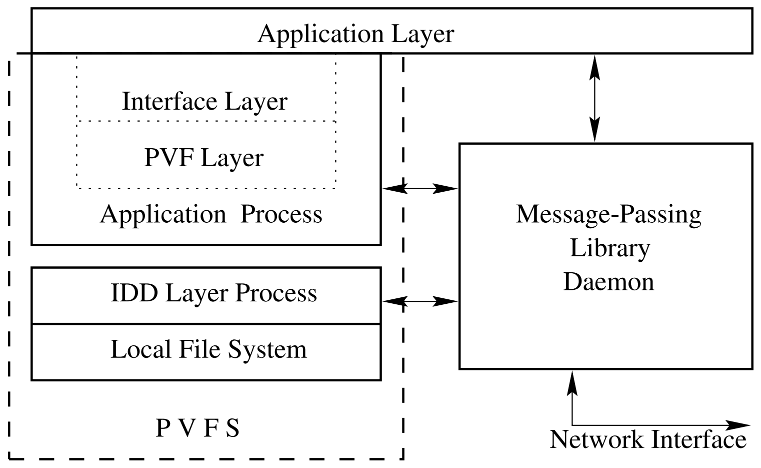


Figure 2.29: VIP-FS Functional Organization

- **I/O device drive layer (IDD)**: built upon and communicates with the local host's file system. It manages each file as an independent non-parallel file and provides a stateless abstraction to the **VPF** layer. Furthermore, the **IDD** layer acts as a mediator between the local host file system and the **VPF** layer (see Figure 2.29).

A configuration file has to include a list of participating hosts, the number of processes to create on each host, some additional path information indicating where the executable file is to be found as well as the file system configuration. The organization of VIP-FS has also a disadvantage. The interprocess communication is increased.

ViPIOS (Vienna Parallel Input-Output System)

abstract:

The ViPIOS represents a mass-storage sub-system for highly I/O intensive scientific applications on massively parallel supercomputers [126]. The ViPIOS is based on a client-server approach combining the advantages of parallel I/O runtime libraries and parallel file systems [18]. I/O bandwidth is maximized by exploiting the concept of data locality, by optimizing the data layout on disk and, thus, allowing efficient parallel read/write accesses. What is more, the ViPIOS is influenced by the concepts of parallel database technology [20].

aims:

The main goal is to provide performance for file I/O requests. The ViPIOS has the following characteristics [126]:

- parallelism (processors access multiple disks in parallel)
- independence (executing in parallel to the applications)
- **scalability**
- abstract I/O model (data independent views)
- modularized interfaces
- portability

implementation platform:

Cluster architectures, MPP, **message passing** systems

data access strategies:

Keeping the principle of data locality, "owner stores" rule, by choosing an appropriate data layout on disk.

portability:

Providing a runtime I/O module for ***HPF*** (Vienna Fortran Compiler VFC), an ***MPI-IO*** interface, and a propriatory ViPIOS interface.

related work:

Galley, Vesta, ***PIOUS***, ***PASSION***, ***Jovian***, ***Panda***

people:

Erich Schikuta (principal investigator), Thomas A. Mück, Peter Brezany, Thomas Fürle, Oliver Jorns, Heinz & Kurt Stockinger, Helmut Wanek

`{schiki, mueck}@ifs.univie.ac.at`, `brezany@par.univie.ac.at`

`{fuerle, jorns, heinz, kurt, wanek}@vipios.pri.univie.ac.at`

institution:

Institute for Applied Computer Science and Information Systems, Department of Data Engineering, University of Vienna, Rathausstr. 19/4, A-1010 Vienna, Austria

`http://vipios.pri.univie.ac.at/`

key words:

high performance mass storage system, high performance language, compilation techniques, data administration

details:

The ViPIOS chooses the file layout as close as possible to the problem specification in focus to reach data local accesses. It is not guaranteed that the physical distribution is equal to the problem distribution. Thus, the physical data layout is transparent to the application processes and provided by different view layers (represented by file pointers accordingly) to the application programmer. A prototype implementation is ready; the performance analysis shows promising results.

The ViPIOS distinguishes between **application process**es and ViPIOS servers, which are similar to data server processes in a database system [19]. Additionally, a ViPIOS supervisor server administrates all other ViPIOS processes. For each **application process** there is exactly one ViPIOS server, but one ViPIOS server can serve a number of **application process**es. The ViPIOS organizes data according to the information provided by **VFCS**.

Data locality is another principle of the ViPIOS, which means that data requested by an **application process** should be read/written from/to the best-suited disk [19]. **Data locality** can further be divided into **physical** and **logical data locality**. Whereas **logical data locality** refers to the best-suited ViPIOS server, the **physical data locality** determines the best disk set. The process model is illustrated in Figure 2.30 where VI is the ViPIOS Interface, and AP is an **application process** handling the communication with the assigned ViPIOS server and linked with the VI.

Data administration is executed using a ***Two-Phase Method***. The first phase is a preparation phase and the second an administration phase. The preparation phase chooses the best suited data layout according to compile time infomation of the application processes and redistributes the data if necessary before the application executes. The administration phase performs the data requests of the application (read and write operations) during execution

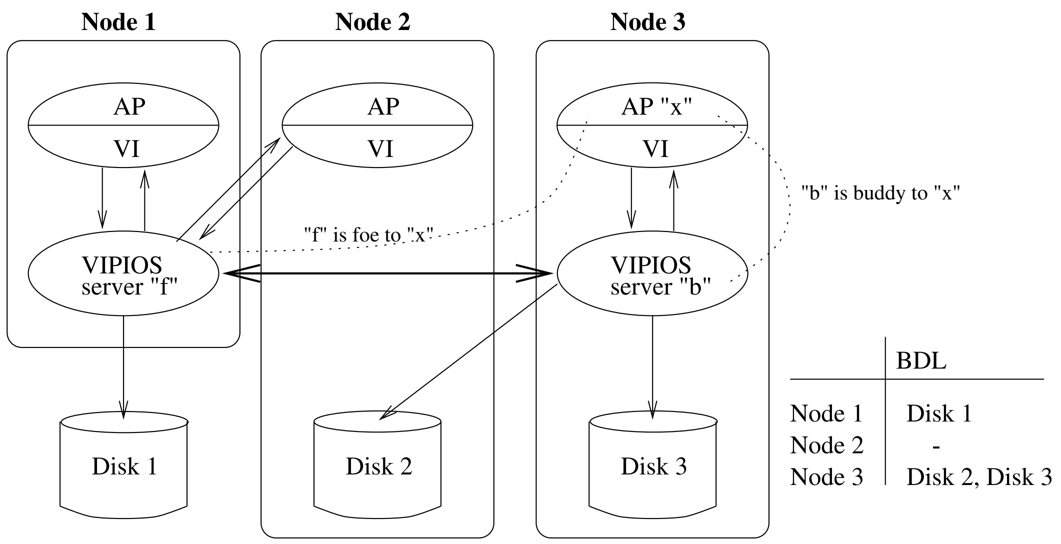


Figure 2.30: Process model of application processes and ViPIOS servers

of the application. The ViPIOS also considers software-controlled ***prefetching*** by using the compile-time knowledge about loops and **OOC** parts and representing it by a graph structure called **I/O Requirement Graph (IORG)**. The **IORG** is denoted by a triple G=(N, E, s) where (N, E) is a directed graph and s is the start node.

virtual machine see ***PVM***
virtual memory see ***design of parallel I/O software, OODB, PASSION***
Virtual Parallel File Layer (VPF) see ***Virtual Parallel File System***

Vulcan multicomputer

This computer located at the IBM T. J. Watson Research Center is a **distributed memory**, **message passing** machine, with nodes connected by a multistage packet-switched network [36]. The nodes include **compute node**s as well as **storage node**s. See also ***Vesta***.

Appendix A

Overview of different parallel I/O products

The aim of this chapter is to compare different approaches and solutions of the various research teams listed in the dictionary part. (A detailed list of reserach teams is illustrated in Table A.2. Annotation: (et. al.) means that the specific product was produced by more than one institution.) The I/O products can be splitted into three different groups:

- file systems (see Figure A.1)
- I/O libraries (see Figure A.2)
- others, i.e. products that are neither file systems nor I/O libraries (see Figures A.3 and A.4)

The platforms which are used by the various approaches are listed in Table A.1.

Figure A.1: Parallel I/O products: Parallel File Systems

name	institution	memory model	sync/async	SPMD/SIMD/MIMD	strided access	data parallel	message passing	client-server	clustering	caching	prefetching	collective operations	shared file pointer	concurrency control	views	implementation platf.	new ideas
CCFS	University of Madrid	DM	+ / +	- / -/ +	-	-	+	+	+	+	+	-	+	+	+	UNIX,24,25	IBL, group operations, automatic preallocation of resources
CFS	Intel	DM	+ /												+	18	four I/O modes
ELFS	University of Virginia	DM								+	+						OO, ease-of-use
Galley	Dartmouth College	DM	+ / +	+ / - / +	+	-	+	+	+	+	-	-	-	-	-	15, UNIX	3d structure of a file
HFS	University of Toronto	SM							+							13, 21	hierarchical clustering, ASF, storage objects
HiDIOS	Australian Nat. University	DM					+			+						11	disk level parallelism
OSF/1	Intel	DM														17	
ParFiSys	University of Madrid	DM	+ / +	- / - / +	-	+	+	+	+	+	+	-	+	+	+	UNIX,15,24,25	IBL, group operations, automatic preallocation of resources
PFS	Intel	DM														17	I/O in parallel wherever possible
PIOFS	IBM	DM													+	13, 15	
PIOUS	Emory University	DM	+ / +	+ / + / +	-	+	+	+	+	-	+	-	+	+	+	23	I/O for metacomputing environment
PPFS	University of Illinois	DM	+ / +		+		+	+	+	+	+					3, 17, 23	
SPFS	Carnegie Mellon University	SM														7, 10	
SPIFFI	University of Wisconsin	DM	+ /								+	+	+			17	three types of threads
Vesta	IBM	DM	+ / +	/ / +			+	+			+	+	-	+	+	12, 14	
VIP-FS	University of Syracuse	DM					+								+	14	three layers

Annotation: + ... "is supported"
- ... "is not supported"
DM ... distributed memory
SM ... shared memory
Numbers listed at "implementation platform" represent machines stated at Table A.1.

Figure A.2: Parallel I/O products: I/O Libraries

name	institution	sync/async	SPMD/SIMD/MIMD	strided access	data parallel	message passing	client-server	clustering	caching	prefetching	collective operations	shared file pointer	concurrency control	views	implementation platf.	new ideas
ADIO	Dartmouth College	+ / +	+ / - / +	+	+	+	-	+	-	-	+	-	+	+	15, 17	strategies for implementing APIs
CVL	Dartmouth College		/ + /												6	vector operations
DDLY	University of Malaga		/ + /			+					+					VDS, IDS
Jovian	University of Maryland	+ /	+ / /			+			+		+			+	14, 17	global, distributed view
MPI	MPI Forum	+ / +	+ / + / +	+		+					+					derived data types, communicators
MPI-2	MPI Forum	+ / +	+ / + / +	+		+					+	+		+		I/O for MPI
MPI-IO	MPI-IO Committee	+ / +	+ / + / +			+					+	+		+		I/O for MPI
Multipol	University of California	+ / +								-					3, 14, 17	PD, distributed data structures
Panda	University of Illinois	+ / +	+ / - / -	+	-	+	+		+		+	-	+	-	15,16, 17	server-directed I/O, chunking, data compression
PASSION	University of Syracuse	+ /	+ / /		+	+			+	+	+				15, 18	TPM, irregular problems
PVM	Oak Ridge National Laboratory, University of Tennesse, Carnegie Mellon University	+ / +	+ / + / +			+	+	+			+					Master-slave, metacomputing
ROMIO	Darmouth College	+ / +	+ / - / +	+	+	+	-	+	-	-	+	-	+	+	UNIX	portable implementation of MPI-IO
TPIE	Duke Uni., Uni. of Delware														8, 23	
ViPIOS	University of Vienna	+ / +	+ / + /	+	+	+	+	+	+	+	+	+		+	15	influence from DB technology

Annotation: + ... "is supported"
- ... "is not supported"
Numbers listed at "implementation platform" represent machines stated at Table A.1.

Figure A.3: Parallel I/O products: others (1)

name	institution	description	new ideas
ADOPT	Syracuse University	prefetching scheme	
Agent Tcl	Dartmouth College	transportable agent	transportable agent, migration
CHANNEL	Syracuse Universiy	communication, synchronization	channls for communiction
CHAOS	University of Maryland	coupling multiple data-parallel programs	mapping between data structures
ChemIO	Scalable I/O Initiative		guidelines for language features, compiler, system support services, high performance network software
disk-directed I/O	Dartmouth College	prefetching scheme	caching, prefetching, collective I/O
EXODUS	University of Wisconsin	OO database	OODB (basis for SHORE)
Global Arrays	Parcific Nortwest Laboratory	interface (combination of DM and SM features)	access of logical blocks on physically distributed machines
Fortran D	Rice University	programming language based on Fortran77	data decomposition
OPT++	University of Wisconsin	OO tool for DB query optimization	
Pablo	University of Illinois	performance analysis tool	SDDF
Paradise	University of Wisconsin	OODB approach of EXODUS	
PARTI	University of Maryland	subset of CHAOS; toolkit	irregular problems
PRE	University of Maryland	code optimization	IPRE (redundancy)
RAPID	Duke University	file system testbed	testbed: buffering and prefetching
Scotch	Carnegie Mellon University	testbed	DAC, TIP, disclosure
SHORE	University of Wisconsin	persistent object system	SDL
TIAS	Dartmouth College	transportable agent	agent, migration
TOPs	University of Maryland	portable software for fast parallel I/O	
ViC*	Dartmouth College	C vectory library; compiler for out-of-core C*	optimizes via loop transformations; includes library of optimal algorithms for the Parallel Disk Model

name	sync/async	SPMD/SIMD/MIMD	strided access	data parallel	message passing	client-server	clustering	caching	prefetching	collective operations	shared file pointer	concurrency control	views	implementation platf.
ADOPT								+	+					
Agent Tcl	- / +	- / -/ -	-	-	+	-	-	-	-	+	-	-	-	UNIX
CHANNEL	+ / ?			+										
CHAOS	? / +			+	+				+					22
disk-directed I/O	+ / -	+ / - / -	+	p	+	+	+	+	+	+	-	-	-	
EXODUS						+								
Global Arrays	+ / +	? / ? / +			+		+			+				14, 17, 18, 19
Fortran D		+ / ? / ?		+										UNIX
OPT++														23
Pablo														16
Paradise														17
PARTI				+				+						16, 18
PRE														16
RAPID	+ / -	+ / - / -	-	+	-	-	-	+	+	-	-	-	-	1
Scotch	- /-						+							7, 10
SHORE				+			+							
TIAS					+	-								8, 9
TOPs						+								15
ViC*	+ / -	+ / - / -	+	+	+	-	-	-	+	+	-	-	-	

Annotation: + ... "is supported"

- ... "is not supported"

p ... possible

Numbers listed at "implementation platform" represent machines stated at Table A.1.

Figure A.4: Parallel I/O products: others (2)

No.	name (usage)	amount of usage
1	Butterfly Plus (RAPID)	1
2	CM-2	0
3	CM-5 (Multipol, PPFS)	2
4	Cray C90	0
5	Cray Y-MP	0
6	DEC 12000/Sx 2000 (CVL)	1
7	DEC 3000/500 (SPFS)	1
8	DEC Alpha (TIAS)	1
9	DEC MIPS (TIAS)	1
10	DEC-5000 (STARFISH, SPFS)	2
11	Fujitsu AP1000 (HiDIOS)	1
12	IBM R6000/350 (HFS, Vesta)	2
13	IBM RS/6000 (PIOFS)	1
14	IBM SP1 (GA, Jovian, Multipol, ROMIO, Vesta, VIP-FS, ViPIOS)	7
15	IBM SP2 (ADIO, DRA, Galley, Panda, PASSION, ParFiSys, PIOFS, ROMIO, SIOF, TOPs, ViPIOS)	11
16	Intel iPSC/860 hypercube (CHARISMA, Pablo, Panda, PARTI, PRE)	5
17	Intel Paragon (ADIO, DRA, GA, Multipol, OSF/1, Panda, Paradise, PFS, PPFS, ROMIO, SPIFFI)	11
18	Intel Touchstone Delta (CFS, GA, PARTI, PASSION)	4
19	Kendal Square KSR-2 (GA)	1
20	MasPar MP-2	0
21	SGI (HFS)	1
22	SMP Digital Alpha (CHAOS)	1
23	SPARC (OPT++, PIOUS, TPIE)	3
24	T800 (ParFiSys)	1
25	T9000 (ParFiSys)	1

Table A.1: The usage of hardware platforms.

Institution	**Product**
Argonne National Laboratories	ADIO, ROMIO
Australian National University	HiDIOS
Carnegie Mellon University	PVM (et. al.), Scotch
Dartmouth College	Agent Tcl, CHARISMA, CVL disk-directed I/O, Galley, RAPID, TIAS, ViC*
Duke University	TPIE
Emory University	PIOUS, PVM (et. al.)
IBM	PIOFS, Vesta
Intel	OFS/1, CFS, PFS
Message Passing Interface Forum	MPI, MPI-2
MPI-IO Committee	MPI-IO
Oak Ridge National Laboratory	PVM (et. al.)
Parcific Northwest Lab.	Global Arrays
Rice University	Fortran D
Scalable I/O Initiative	ChemIO
University of California	Mulipol, RAID, raidPerf, raidSim, SIOF
University of Delware	TPIE (et. al.)
University of Illinois	Pablo, Panda, PPFS
University of Madrid	CCFS, ParFiSys
University of Malaga	DDLY
University of Maryland	CHAOS, Jovian, PARTI, PRE, TOPs
University of Syracuse	ADOPT, CHANNEL, PASSION, Two-Phase Method, VIP-FS
University of Tennessee	PVM (et. al.)
University of Toronto	HFS, Hector
University of Vienna	Vienna Fortran, ViPIOS
University of Virginia	ELFS
University of Wisconsin	EXODUS, OPT++, Paradise, SHORE, SPIFFI

Table A.2: Research teams and their products.

Appendix B

Parallel Computer Architectures

Since there appear so many different machines in this dictionary, this part of the appendix shall give an overview of parallel architectures in general. Furthermore, some of the machines mentioned in the dictionary part will be explained explicitly and in more detail.

In general, there are two main architectures for parallel machines, SIMD and MIMD architectures. SIMD machines are supposed to be the cheapest ones, and the architecture is not as complex as in MIMD machines. In particular, all the processing elements have to execute the same instructions whereas in MIMD machines many programs can be executed at the same time. Hence, they are said to be the ”real” parallel machines [69]. A major difference between the two types is the interconnecting network. In a SIMD architecture this network is a static one while MIMD machines have different ones depending on the organization of the address space. This also results in two different communication mechanisms for MIMD machines: message passing systems (also called distributed memory machines) and virtual shared memory systems (NUMA: nonuniform memory access). Massively parallel machines apply UMA architectures that are based on special crossbar interconnecting networks.

SIMD Machines

These machines are supposed to be ”out of date” [69], but they are still in use.

- **Connection Machines CM-200**

  This machine was built by the Thinking Machines Corporation. CM-200 is the most modern edition of version 2 (CM-2). The machine consists of 4096 to 65535 microprocessors with a one-bit word length. Moreover, the bit-architecture of each processing element enables to define different instructions. In comparison, CM-5 is a MIMD machine.

- **MasPar-2**

  This machine was built by MasPar, an affiliate company from DEC. MasPar-2 consists of up to 16K 32-bit micro processors. Although float comma operations are micro coded, the performance for float comma operations of CM-200 is better. Furthermore, the front-end computer is a DECstation 5000.

Distributed Memory MIMD Machines

One processor can only directly access its own memory while the memories of other processors have to be accessed via message passing.

MIMD machines have some different topologies like hypercube or grid. The interconnectivity depends on the amount of links and whether they can be used concurrently. Systems with a flat topology need four links in order to establish a 2-dimensional grid (Intel Paragon) whereas systems with 3-d grids need six links. Hypercubes need most links, and each processing node has a links to neighboring nodes.

- **Hypercube Systems**

  - **Intel iPSC/860**: 60 MFLOPS (60 MHz)
  - **nCUBE-2S**: 15 MIPS, 4 MFLOPS
  - **nCUBE-3**: 50 MIPS, 50 MFLOPS (floating point), 100 MFLOPS (”multiply-add”)

- **2-dimensional Topologies**
  - **INMOS Transputer T805**: 5 MIPS, 1,5 MFLOPS
  - **Intel Paragon XP/S**: (max. 300 GFLOPS) 50 MHz i860XP: 40 MIPS, 75 MFLOPS
- **3-dimensional Topologies**
  - **Parsytec GC** (based on an INMOS T9000 processor): 25 MFLOPS
- **Multilevel Interconnecting Network**
  - **Thinking Machines CM-5**

    The principles of CM-2/CM-200 and the MIMD principle are combined in the CM-5, and the processors are normal SPARC micro processors. 4*32 MFLOPS per node
  - **IBM SP2**

    The SP2 consists of RS/6000 processors. 125 MFLOPS (thin nodes), 266 MFLOPS (wide nodes)

Shared Memory MIMD Machines

In contrast to a Cray Y-MP where a uniform memory access is used, in shared memory machines the amount of processors is much bigger and, hence, non-uniform memory access is applied.

- **Ring Topologies**
  - **Kendal Square Research KSR-2 AllCache**: 80 MIPS, 80 MFLOPS Convex Exemplar SPP1000: 200 MFLOPS
- **3-dimensional Topologies**
  - **CRAY T3D**

    This is a massive parallel machine with up to 2048 processors. 150 MFLOPS

- **Multilevel Interconnecting Network**
  - **MANNA**

    MANNA is a massively parallel machine for numeric and non-numeric applications. 50 MHz, 50 MIPS, 100 MFLOPS (32 bit), 50 MFLOPS (64 bit)
  - **Meiko CS-2**

    Fujitsu vector multiprocessor: 100 MFLOPS, SPARC RISC processor: 40 MFLOPS 4 Crossbar switch
  - **Fujitsu VPP500**

    This machine is supposed to be one of the most powerful massively parallel systems. 1,6 GFLOPS (vectors), 300 MIPS and 200 MFLOPS (scalars)

Bibliography

[1] Vikram S. Adve, John Mellor-Crummey, Mark Anderson, Ken Kennedy, Jhy-Chun Wang, and Daniel A. Reed. An integrated compilation and performace analysis environment for data parallel programs. Center for Research on Parallel Computation, Rice University, University of Illinois at Urbana-Champaign, Department of Computer Science.

[2] Gagan Agrawal, Joel Saltz, and Raja Das. Interprocedural partial redundancy elimination and its application to distributed memory compilation. Department of Computer Science and UMICAS, University of Maryland.

[3] Georgae S. Alamasi and Allan Gottlieb. *Highly Parallel Computing.* Benjamin/Cummings Publishing Company, 1989.

[4] Bhavan Avalani, Alok Choudhary, Ian Foster, and Rakesh Kirshnaiyer. Integrating task and data parallelism using parallel I/O techniques. In *Proceedings of the International Workshop on Parallel Processing*, Bangalore, India, December 1994.

[5] Ruth A. Aydt. The pablo self-defining data format, March 1992. University of Illinois at Urbana-Champaign, Department of Computer Science.

[6] Rajive Bagrodia, Andrew Chien, Yarson Hsu, and Daniel Reed. Input/output: Instrumentation, characterization, modeling and management policy. Technical Report CCSF-41, Scalable I/O Initiative, Caltech Concurrent Supercomputing Facilities, Caltech, 1994.

[7] Mary J. Baker, John H. Hartmann, Michael D. Kupfer, Ken W Shirriff, and John K. Ousterhout. Measurements of a distributed file system, 1991. Electrical Engineering and Computer Science Division, University of California, Berkeley.

[8] Sandra Johnson Baylor and C. Eric Wu. Parallel I/O workload characteristics using Vesta. In *Proceedings of the IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 16–29, April 1995.

[9] Robert Bennett, Kelvin Bryant, Alan Sussman, Raja Das, and Joel Saltz. Jovian: A framework for optimizing parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 10–20, Mississippi State, MS, October 1994. IEEE Computer Society Press.

[10] Brian Bershad, David Black, David DeWitt, Garth Gibson, Kai Li, Larry Peterson, and Marc Snir. Operating system support for high-performance parallel I/O systems. Technical Report CCSF-40, Scalable I/O Initiative, Caltech Concurrent Supercomputing Facilities, Caltech, 1994.

[11] Rajesh Bordawekar and Alok Choudhary. Communication strategies for out-of-core programs on distributed memory machines. Technical Report SCCS-667, NPAC, Syracuse University, 1994.

[12] Rajesh Bordawekar and Alok Choudhary. Compiler and runtime support for parallel I/O. In *Proceedings of IFIP Working Conference (WG10.3) on Programming Environments for Massively Parallel Distributed Systems*, Monte Verita, Ascona, Switzerland, April 1994. Birkhaeuser Verlag AG, Basel, Switzerland.

[13] Rajesh Bordawekar, Alok Choudhary, and J. Ramanujam. Automatic optimization of communication in out-of-core stencil codes. Technical Report CACR-114, Scalable I/O Initiative, Center of Advanced Computing Research, California Insititute of Technology, November 1995.

[14] Rajesh Bordawekar, Alok Choudhary, and J. Ramanujam. Compilation and communication strategies for out-of-core programs on distributed-memory machines. *Journal of Parallel and Distributed Computing*, 38(2):277–288, November 1996.

[15] Rajesh Bordawekar, Alok Choudhary, and Juan Miguel Del Rosario. An experimental performance evaluation of Touchstone Delta Concurrent File System. In *Proceedings of the 7th ACM International Conference on Supercomputing*, pages 367–376. ACM Press, 1993.

[16] Rajesh Bordawekar, Juan Miguel del Rosario, and Alok Choudhary. Design and evaluation of primitives for parallel I/O. In *Proceedings of Supercomputing '93*, pages 452–461, Portland, OR, 1993. IEEE Computer Society Press.

[17] P. Brezany and A. Choudhary. Techniques and optimizations for developing irregular out-of-core applications on distributed-memory systems. Technical Report 96-4, Institute for Software Technology and Parallel Systems, University of Vienna, November 1996.

[18] Peter Brezany, Thomas A. Mueck, and Erich Schikuta. Language, compiler and parallel database support for I/O intensive applications. In *Proceedings of the International Conference on High Performance Computing and Networking*, volume 919 of *Lecture Notes in Computer Science*, pages 14–20, Milan, Italy, May 1995. Springer-Verlag. also available as Technical Report of the Inst. f. Software Technology and Parallel Systems, University of Vienna, TR95-8, 1995.

[19] Peter Brezany, Thomas A. Mueck, and Erich Schikuta. Mass storage support for a parallelizing compilation system. In *International Conference Eurosim'96– HPCN challenges in Telecomp and Telecom: Parallel Simulation of Complex Systems and Large Scale Applications*, pages 63–70, Delft, The Netherlands, June 1996. North-Holland, Elsevier Science.

[20] Peter Brezany, Thomas A. Mueck, and Erich Schikuta. A software architecture for massively parallel input-output. In *Third International Workshop PARA'96 (Applied Parallel Computing - Industrial Computation and Optimization)*, volume 1186 of *Lecture Notes in Computer Science*, pages 85–96, Lyngby, Denmark, August 1996. Springer-Verlag. Also available as Technical Report of the Inst. f. Angewandte Informatik u. Informationssysteme, University of Vienna, TR 96202.

[21] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, Seth J. White, and Michael J. Zwilling. Shoring up persistent applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 383–394. ACM Press, 1994.

[22] Michael J. Carey, Michael J. Franklin, and Markos Zaharioudakis. Fine-grained sharing in a page server oodbms. Computer Science Department, University of Wisconsin, Madison.

[23] J. Carretero, F. Pérez, P. de Miguel, F. García, and L. Alonso. A multiprocessor parallel disk system evaluation. *Decentralized and Distributed Systems*, September 1993. IFIP Transactions A-39.

[24] J. Carretero, F. Pérez, P. de Miguel, F. García, and L. Alonso. Implementation of a parallel file system: CCFS a case of study. Technical Report FIM/84.1/DATSI/94, Universidad Politecnic Madrid, Madrid, Spain, 1994.

[25] J. Carretero, F. Pérez, P. de Miguel, F. García, and L. Alonso. POSIX-style parallel file server for the GPMIMD: Final report. Technical Report D1.7/2, Universidad Politecnic Madrid, Madrid, Spain, 1995.

[26] J. Carretero, F. Pérez, P. de Miguel, F. García, and L. Alonso. I/O data mapping in *ParFiSys:* support for high-performance I/O in parallel and distributed systems. In *Euro-Par '96*, volume 1123 of *Lecture Notes in Computer Science*, pages 522–526. Springer-Verlag, August 1996.

[27] J. Carretero, F. Pérez, P. de Miguel, F. García, and L. Alonso. ParFiSys: A parallel file system for MPP. *ACM Operating Systems Review*, 30(2):74–80, April 1996.

[28] J. Carretero, F. Pérez, P. de Miguel, F. García, and L. Alonso. Performance increase mechanisms for parallel and distributed file systems. *Parallel Computing*, 23(4):525–542, June 1997.

[29] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[30] Y. Chen, M. Winslett, K. E. Seamons, S. Kuo, Y. Cho, and M. Subramaniam. Scalable message passing in Panda. In *Proceedings of the Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 109–121, Philadelphia, May 1996. ACM Press.

[31] A Choudhary, B. Bordawekar, A. Dalia, S. More, K. Sivaram, and R. Thakur. *A User's Guide for the PASSION Runtime Library Version 1.1.* Syracuse University, October 1995.

[32] Alok Choudhary, Rajesh Bordawekar, Michael Harry, Rakesh Krishnaiyer, Ravi Ponnusamy, Tarvinder Singh, and Rajeev Thakur. PASSION: parallel and scalable software for input-output. Technical Report SCCS-636, ECE Dept., NPAC and CASE Center, Syracuse University, September 1994.

[33] Alok Choudhary, Rajesh Bordawekar, Sachin More, K. Sivaram, and Rajeev Thakur. PASSION runtime library for the Intel Paragon. In *Proceedings of the Intel Supercomputer User's Group Conference*, June 1995.

[34] Alok Choudhary, Ian Foster, Geoffrey Fox, Ken Kennedy, Carl Kesselman, Charles Koelbel, Joel Saltz, and Marc Snir. Languages, compilers, and runtime systems support for parallel input-output. Technical Report CCSF-39, Scalable I/O Initiative, Caltech Concurrent Supercomputing Facilities, Caltech, 1994.

[35] Alex Colvin and Thomas H. Cormen. Vic*: A compiler for virtual-memory c*. Technical Report PCS-TR97-323, Dartmouth College, Computer Science, Hanover, NH, November 1997.

[36] Peter F. Corbett and Dror G. Feitelson. Design and implementation of the Vesta parallel file system. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 63–70, 1994.

[37] Peter F. Corbett, Dror G. Feitelson, Jean-Pierre Prost, and Sandra Johnson Baylor. Parallel access to files in the Vesta file system. In *Proceedings of Supercomputing '93*, pages 472–481, Portland, OR, 1993. IEEE Computer Society Press.

[38] Thomas H. Cormen, Sumit Chawla, Preston Crow, Melissa Hirschl, Roberto Hoyle, Keith D. Kotay, Rolf H. Nelson, Nils Nieuwejaar, Scott M. Silver, Michael B. Taylor, and Rajiv Wickremesinghe. Dartcvl: The darmouth c vektor library. Technical report, 1995. PCS-TR95-250.

[39] Thomas H. Cormen and Alex Colvin. ViC*: A preprocessor for virtual-memory C*. Technical Report PCS-TR94-243, Dept. of Computer Science, Dartmouth College, November 1994.

[40] Thomas H. Cormen and David Kotz. Integrating theory and practice in parallel file systems. Technical Report PCS-TR93-188, Dept. of Math and Computer Science, Dartmouth College, March 1993. Revised 9/20/94.

[41] Thomas H. Cormen and Leonard F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. In *Proceedings of the Fifth Symposium on Parallel Algorithms and Architectures*, pages 130–139, June 1993.

[42] William V. Courtright, Garth Gibson, Mark Hollenad, and Jim Zelenka. A structured approach to redundant disk array implementation. Department of Electrical Engineering and Computer Engineering, School of Computer Science, Carnegie Mellon University.

[43] Raja Das, Yuan-Shin Hwang, Mustafa Uysal, Joel Saltz, and Alan Sussman. Applying the chaos/parti library to irregular problems in computational chemistry and computational aerodynamics. Department of Computer Science, University of Maryland.

[44] Raja Das, Joel Saltz, and Reinhard von Hanxleden. Slicing analysis and indirect access to distributed arrays. Technical report, June 1993. CRPC-TR 93319-S.

[45] Raja Das, Mustafa Uysal, Joel Saltz, and Yuan-Shin Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. In *Journal of Parallel and Distributed Computing*, September 1994.

[46] Juan Miguel del Rosario, Michael Harry, and Alok Choudhary. The design of VIP-FS: A virtual, parallel file system for high performance parallel and distributed computing. Technical Report SCCS-628, NPAC, Syracuse, NY 13244, May 1994.

[47] David J. DeWitt, Navin Kabra, JunLuo, Jignesh M. Patel, and Jie-Bing Yu. Client-server paradise. 1994. Santiago, Chile.

[48] David J. DeWitt, Jeffrey Naughton, John C. Shafer, and Shivakumar Venkataraman. Parsets for parallelizing oodbms traversals: Implemenation and performance. Computer Science Department, University of Wisconsin, Madison.

[49] Guy Edjlali, Alan Sussman, and Joel Saltz. Interoperability of data parallel runtime libraries with meta-chaos. Department of Computer Science, University of Maryland.

[50] Chris Elford, Chris Kuszmaul, Jay Huber, and Tara Madhyastha. Design of a portable parallel file system. Technical report, November 1993.

[51] Chris Elford, Chris Kuszmaul, Jay Huber, and Tara Madhyastha. Portable parallel file system detailed design. Technical report, University of Illinois at Urbana-Champaign, November 1993.

[52] Message Passing Interface Forum. Mpi: A message-passing interface standard, June 1994. University of Tennessee.

[53] Michael Franklin, Michael J. Carey, and Miron Livny. Global memory management in client server dbms architectures. Computer Science Department, University of Wisconsin, Madison.

[54] Craig S. Freedman, Josef Burger, and David J. Dewitt. SPIFFI — a scalable parallel file system for the Intel Paragon. *IEEE Transactions on Parallel and Distributed Systems*, 7(11):1185–1200, November 1996.

[55] Al Geist, Adan Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. Pvm: Parallel virtual machine - a user's guide and tutorial for networked parallel computing. MIT Press, 1994.

[56] Al Geist and et. al. Pvm and mpi: A comparison of features, 1996. http://www.netlib.org/pvm3/.

[57] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David Rochberg, and Jim Zelenka. File server scaling with network-attached secure disks. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (Sigmetrics '97)*, June 1997. Washington.

[58] Garth A. Gibson, R. Hugo Patterson, and M. Satyanarayanan. Disk reads with DRAM latency. In *Third Workshop on Workstation Operating Systems*, pages 126–131, 1992.

[59] Garth A. Gibson, Daniel Stodolsky, Pay W. Chang, William V. Courtright II, Chris G. Demetriou, Eka Ginting, Mark Holland, Qingming Ma, LeAnn Neal, R. Hugo Patterson, Jiawen Su, Rachad Youssef, and Jim Zelenka. The Scotch parallel storage systems. In *Proceedings of 40th IEEE Computer Society International Conference (COMPCON 95)*, pages 403–410, San Francisco, Spring 1995.

[60] Robert Gray, George Cybenko, David Kotz, and Daniela Rus. Agent tcl. Technical report, 1996.

[61] Robert S. Gray. Agent tcl: A transportable agent system. In James Mayfield and Tim Finin, editors, *Proceedings of the CIKM Workshop on Intelligent Information Agents, Forth International Conference on Information and Knowledge (CIKM 95)*, December 1995. Baltimore, Maryland.

[62] Robert S. Gray. *Transportable Agents*. PhD thesis, Department of Computer Science, Dartmouth College, May 1995.

[63] Robert S. Gray. A flexible and secure mobile-agent system. In Mark Diekhans and Mark Roseman, editors, *Proceedings of the Fourth Annual Tcl/Tk Workshop (TCL '96)*, July 1996. Monterey, California.

[64] The Pablo Research Group. A description of the classes and methods of the pablo sddf interface library, May 1993. University of Illinois at Urbana-Champaign, Department of Computer Science.

[65] Kenneth E. Harker. Tias: A transportable intelligent agent system. Technical report, 1995.

[66] Michael Harry, Juan Miguel del Rosario, and Alok Choudhary. VIP-FS: A VIrtual, Parallel File System for high performance parallel and distributed computing. In *Proceedings of the Ninth International Parallel Processing Symposium*, pages 159–164, April 1995. Also appeared in ACM Operating Systems Review 29(3), July 1995 pages 35–48.

[67] Lisa Helerstein, . Gibson Garth A, Richard M. Karp, Randy H. Kth, and David A. Patterson. Coding techniques for handling failures in large disk arrays. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, March 1989. Boston.

[68] Mark Henderson, Bill Nickless, and Rick Stevens. A scalable high-performance I/O system. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 79–86, 1994.

[69] Friedrich Hertweck. *Parallelrechner*, chapter Realisierung paralleler Architekturen. B.G. Teubner Stuttgard, 1995. Klaus Waldschmidt (Hrsg.).

[70] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Preliminary experiences with the fortran d compiler. Technical report, April 1993. CRPC-TR 93397.

[71] Jay Huber, Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal. PPFS: A high performance portable parallel file system. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394, Barcelona, July 1995. ACM Press.

[72] Jay Huber, Chris Kuszmaul, Tara Madhyastha, and Chris Elford. Scenarios for the portable parallel file system. Technical report, University of Illinois at Urbana-Champaign, November 1993.

[73] James V. Huber, Jr. PPFS: An experimental file system for high performance parallel input/output. Master's thesis, Department of Computer Science, University of Illinois at Urbana Champaign, February 1995.

[74] IBM. Parallel i/o file system (piofs) for aix, 1997. http://www.rs6000.ibm.rescouces.sp_books/piofs/index.html.

[75] Intel. Intel paragon supercomputers, 1997. http://www.greed.isc.tamu.edu/paragon /user_guide/ch5.html.

[76] Navin Kabra and David J. DeWitt. Opt++: An object-oriented implementation for extensibe database query optimization. Computer Science Department, University of Wisconsin, Madison.

[77] John F. Karpovich, James C. French, and Andrew S. Grimshaw. High performance access to radio astronomy data: A case study. In *Proceedings of the 7th International Working Conference on Scientific and Statistical Database Management*, September 1994. Also available as Univ. of Virginia TR CS-94-25.

[78] John F. Karpovich, Andrew S. Grimshaw, and James C. French. Breaking the I/O bottleneck at the National Radio Astronomy Observatory (NRAO). Technical Report CS-94-37, University of Virginia, August 1993.

[79] John F. Karpovich, Andrew S. Grimshaw, and James C. French. Extensible file systems (ELFS): An object-oriented approach to high performance file I/O. In *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 191–204, Portland, OR, October 1994. ACM Press.

[80] Ken Kennedy and Ulrich Kremer. Initial framework for automatic data layout in fortran d: A short update on a case study. Technical report, July 1993. CRPC-TR 93324-S.

[81] Ken Kennedy and Gerald Roth. Context optimization for simd execution. Technical report, April 1993. CRPC-TR 93306-S.

[82] Tracy Kimbrel, Andrew Tomkins, R. Hugo Patterson, Brian Bershad, Pei Cao, Edward Felten, Garth Gibson, Anna R. Karlin, and Kai Li. A trace-driven comparison of algorithms for parallel prefetching and caching. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*, pages 19–34. USENIX Association, October 1996.

[83] David Kotz. Multiprocessor file system interfaces. Technical Report PCS-TR92-179, Dept. of Math and Computer Science, Dartmouth College, March 1992. Revised version appeared in PDIS'93.

[84] David Kotz. Throughput of existing multiprocessor file systems. Technical Report PCS-TR93-190, Dept. of Math and Computer Science, Dartmouth College, May 1993.

[85] David Kotz. Disk-directed I/O for an out-of-core computation. Technical Report PCS-TR95-251, Dept. of Computer Science, Dartmouth College, January 1995.

[86] David Kotz. Expanding the potential for disk-directed I/O. Technical Report PCS-TR95-254, Dept. of Computer Science, Dartmouth College, March 1995.

[87] David Kotz. Interfaces for disk-directed I/O. Technical Report PCS-TR95-270, Dept. of Computer Science, Dartmouth College, September 1995.

[88] David Kotz and Ting Cai. Exploring the use of I/O nodes for computation in a MIMD multiprocessor. Technical Report PCS-TR94-232, Dept. of Computer Science, Dartmouth College, October 1994. Revised 2/20/95.

[89] David Kotz and Nils Nieuwejaar. Dynamic file-access characteristics of a production parallel scientific workload. In *Proceedings of Supercomputing '94*, pages 640–649, Washington, DC, November 1994. IEEE Computer Society Press.

[90] David F. Kotz and Carla Schlatter Ellis. Prefetching in file systems for MIMD multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):218–230, April 1990.

[91] Ulrich Kremer. Automatic data layout for distributed-memory machines. Technical report, February 1993. CRPC-TR93299-S.

[92] Ulrich Kremer, John Mellor-Crumley, Ken Kennedy, and Alan Carle. Automatic data layout for distributed-memory machines in the d programming environment. Technical report, February 1993. CRPC-TR 93298-S.

[93] Orran Krieger. *HFS: A flexible file system for shared-memory multiprocessors.* PhD thesis, University of Toronto, October 1994.

[94] Orran Krieger, Karren Reid, and Michael Stumm. Exploiting mapped files for parallel i/o. In *SPDP Workshop on Modeling and Specification of I/O*, October 1995.

[95] Orran Krieger and Michael Stumm. HFS: a flexible file system for large-scale multiprocessors. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 6–14, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies.

[96] Orran Krieger, Michael Stumm, and Ronald Unrau. The Alloc Stream Facility: A redesign of application-level stream I/O. Technical Report CSRI-275, Computer Systems Research Institute, University of Toronto, Toronto, Canada, M5S 1A1, October 1992.

[97] Arvind Krishnamurthy, Steven Lumetta, David E. Culler, and Katherine Yelick. Connected components on distributed memory machines. University of California at Berkeley.

[98] S. Kuo, M. Winslett, K. Seamons, Y. Chen, Y. Cho, and M. Subramaniam. Application experience with parallel input/output: Panda and the h3expresso black hole simulation on the sp2. University of Illinois at Urbana-Champaign, Department of Computer Science.

[99] Edward K. Lee. Software and performance issues in the implementation of a RAID prototype. Technical Report UCB/CSD 90/573, EECS, Univ. California at Berkeley, May 1990.

[100] Edward K. Lee, Peter M. Chen, John H. Hartmann, Ann L. Chervenak Drapeau, Ethan L. Miller, Randy H. Katz, Garth A. Gibson, and David A. Patterson. Raid-ii: A scalable storgae architecture for high-bandwidth network file service.

[101] Edward K. Lee and Randy H. Katz. An analytic performance model of disk arrays. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 98–109, 1993.

[102] Edward K. Lee and Randy H. Katz. The performance of parity placements in disk arrays. *IEEE Transactions on Computers*, 42(6):651–664, June 1993.

[103] Edward Kihyen Lee. Performance modeling and analysis of disk arrays. University of California at Berkeley.

[104] Message-Passing Interface Forum. *MPI-2.0: Extensions to the Message-Passing Interface*, chapter 9. MPI Forum, June 1997.

[105] Ethan L. Miller. File miragtion on the cray y-mp at the national center for athmospheric research, June 1991. University of California at Berkeley.

[106] Ethan L. Miller and Randy H. Katz. Input/output behavior of supercomputer applications. In *Proceedings of Supercomputing '91*, pages 567–576, Albuquerque, NM, November 1991. IEEE Computer Society Press.

[107] S. Moyer and V. S. Sunderam. Parallel I/O as a parallel application. *International Journal of Supercomputer Applications*, 9(2):95–107, Summer 1995.

[108] Steven A. Moyer and V. S. Sunderam. Characterizing concurrency control performance for the PIOUS parallel file system. Technical Report CSTR-950601, Emory University, June 1995.

[109] Steven A. Moyer and V. S. Sunderam. *PIOUS for PVM Version 1.2 User's Guide and Reference*. Department of Mathematics and Computer Science, Emory University, January 1995.

[110] Steven A. Moyer and V. S. Sunderam. Scalable concurrency control for parallel file systems. Technical Report CSTR-950202, Emory University, February 1995.

[111] MPI-IO: a parallel file I/O interface for MPI. The MPI-IO Committee, April 1996. Version 0.5. See WWW http://lovelace.nas.nasa.gov/MPI-IO/mpi-io-report.0.5.ps.

[112] Jaroslav Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global arrays: A portable 'shared-memory' programming model for distributed memory computers. Pacific Northwest Laboratory, Richland.

[113] Nils Nieuwejaar and David Kotz. Low-level interfaces for high-level parallel I/O. In *Proceedings of the IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 47–62, April 1995.

[114] Nils Nieuwejaar and David Kotz. The Galley parallel file system. In *Proceedings of the 10th ACM International Conference on Supercomputing*, pages 374–381, Philadelphia, PA, May 1996. ACM Press.

[115] Nils Nieuwejaar and David Kotz. Performance of the Galley parallel file system. In *Proceedings of the Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 83–94, Philadelphia, May 1996. ACM Press.

[116] R. H. Patterson, G. A. Gibson, and M. Satyanarayanan. Using transparent informed prefetching to reduce file read latency. In *Proceedings of the 1992 NASA Goddard conference on Mass Storage Systems*, pages 329–342, September 1992.

[117] R. Hugo Patterson and Garth A. Gibson. Exposing I/O concurrency with informed prefetching. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, pages 7–16, September 1994.

[118] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 79–95, Copper Mountain, CO, December 1995. ACM Press.

[119] R. Hugo Patterson, Garth A. Gibson, and M. Satyanarayanan. A status report on research in transparent informed prefetching. *ACM Operating Systems Review*, 27(2):21–34, April 1993.

[120] Ravi Ponnusamy, Joel Saltz, Alok Choudhary, Yuan-Shin Hwang, and Geoffrey Fox. Runtime support and compilation methods for user-specified irregular data distributions.

[121] James T. Poole. Preliminary survey of I/O intensive applications. Technical Report CCSF-38, Scalable I/O Initiative, Caltech Concurrent Supercomputing Facilities, Caltech, 1994.

[122] Apratim Purakayastha, Carla Schlatter Ellis, David Kotz, Nils Nieuwejaar, and Michael Best. Characterizing parallel file-access patterns on a large-scale multiprocessor. Technical Report CS-1994-33, Dept. of Computer Science, Duke University, October 1994.

[123] M. Ranganathan, A. Acharaya, G. Edjlali, A. Sussman, and J. Saltz. Runtime coupling of data-parallel programs. In ACM Press, editor, *Proceedings of the 1996 International Conference on Supercomputing*, May 1996.

[124] Daniel A. Reed, Ruth A. Aydt, Tara M. Madhyastha, Roger J. Noe, Keith A. Shields, and Bradley W. Schartz. An overview of the pablo performance analysis environment, November 1992. University of Illinois at Urbana-Champaign, Department of Computer Science.

[125] F. Rosales, J. Carretero, F. Pérez, P. de Miguel, F. García, and L. Alonso. CDS design: A parallel disk server for multicomputers. Technical Report FIM/83.1/DATSI/94, Universidad Politecnic Madrid, Madrid, Spain, 1994.

[126] Erich Schikuta. Language, compiler and advanced data structure support for parallel i/o operations, April 1995. Department of Data Engineering, University of Vienna.

[127] David Schneider. Application i/o and related issues on the sp2.

[128] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995. IEEE Computer Society Press.

[129] K. E. Seamons and M. Winslett. An efficient abstract interface for multidimensional array I/O. In *Proceedings of Supercomputing '94*, pages 650–659, Washington, DC, November 1994. IEEE Computer Society Press.

[130] K. E. Seamons and M. Winslett. Physical schemas for large multidimensional arrays in scientific computing applications. In *Proceedings of the 7th International Working Conference on Scientific and Statistical Database Management*, pages 218–227, September 1994.

[131] K. E. Seamons and M. Winslett. A data management approach for handling large compressed arrays in high performance computing. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 119–128, February 1995.

[132] Kent E. Seamons. *Panda: Fast Access to Persistent Arrays Using High Level Interfaces and Server Directed Input/Output.* PhD thesis, University of Illinois at Urbana-Champaign, May 1996.

[133] Tarvinder Pal Singh and Alok Choudhary. ADOPT: A dynamic scheme for optimal prefetching in parallel file systems. Technical report, NPAC, June 1994.

[134] SIOF. Livermore computing: The scalable i/o facility, 1997. http://www.llnl.gov/liv_comp/siof.html.

[135] Heinz Stockinger. Mpi (message passing interface) - a method of parallel computing demonstrated by message passing sorting algorithms, April 1997. Half Unit Project, Department of Computer Science, Royal Holloway College, University of London.

[136] Kurt Stockinger. A distributed genetic algorithm implemented with pvm, April 1997. Half Unit Project, Department of Computer Science, Royal Holloway College, University of London.

[137] Daniel Stodolsky, Mark Holland, William V. Courtright II, and Garth A. Gibson. A redundant disk array architecture for efficient small writes. Technical Report CMU-CS-94-170, Carnegie Mellon University, July 1994. Revised from CMU-CS-93-200.

[138] Mahesh Subramaniam. High performance implementation of server directed i/o. Master's thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, 1996.

[139] Alan Sussman, Joel Saltz, Raja Das, S Gupta, Dimitri Mavriplis, and Ravi Ponnusamy. Parti primitives for unstructured and block strucktured problems.

[140] Michael Tan, Nick Roussopoulos, and Steve Kelley. The Tower of Pizzas. Technical Report UMIACS-TR-95-52, University of Maryland Institute for Advanced Computer Studies (UMIACS), April 1995.

[141] R. Thakur, R. Bordawekar, and A. Choudhary. Compiler and Runtime Support for Out-of-Core HPF Programs. In *Proceedings of the 8th ACM International Conference on Supercomputing*, pages 382–391, Manchester, UK, July 1994. ACM Press.

[142] Rajeev Thakur and Alok Choudhary. Efficient algorithms for array redistribution. Technical report, June 1994. NPAC Technical Report SCCS-601.

[143] Rajeev Thakur and Alok Choudhary. Accessing sections of out-of-core arrays using an extended two-phase method. Technical Report SCCS-685, NPAC, Syracuse University, January 1995.

[144] Rajeev Thakur, William Gropp, and Ewing Lusk. An abstract-device interface for implementing portable parallel-I/O interfaces. In *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 180–187, October 1996.

[145] Rajeev Thakur, Ewing Lusk, and William Gropp. Users guide for ROMIO: A high-performance, portable MPI-IO implementation. Technical Report ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, October 1997.

[146] Andrew Tomkins, R. Hugo Patterson, and Garth Gibson. Informed multi-process prefetching and caching. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (Sigmetrics '97)*, June 1997. Washington.

[147] Guillermo P. Trabado and E. L. Zapata. Support for massive data input/output on parallel computers. In *Proceedings of the Fifth Workshop on Compilers for Parallel Computers*, pages 347–356, June 1995.

[148] Andrew Tridgell and David Walsh. The HiDIOS filesystem. In *Parallel Computing Workshop*, England, September 1995.

[149] Darren Erik Vengroff. {TPIE} user manual and reference, January 1995. Alpha release.

[150] Seth John White. *Pointer Swizzling Techniques for Object-Oriented Database Systems.* PhD thesis, University of Wisconsin - Madison, 1994.

[151] Janet L. Wiener and Jeffrey F. Naughton. Oodb bulk loadig revisited: The partitioned-list apporach. Computer Science Department, University of Wisconsin, Madison.

[152] Katherine Yelick, Soumen Chakrabarti, Etienne Deprit, Jeff Jones, Arvind Krishnamurthy, and Chih po Wen. Data structures for irregular applications. Computer Science Division, University of California, Berkeley.

[153] Katherine Yelick, Soumen Chakrabarti, Etienne Deprit, Jeff Jones, Arvind Krishnamurthy, and Chih po Wen. Parallel data structures for symbolic computation, 1995. Computer Science Division, University of California, Berkeley.

[154] Katherine Yelick, Soumen Chakrabarti, Etienne Deprit, Arvind Krishnamurthy, and Chih po Wen. Runtime support for portable distributed data structures. Computer Science Division, University of California, Berkeley.

[155] Rachad Youssef. RAID for mobile computers. Master's thesis, Carnegie Mellon University Information Networking Institute, August 1995. Available as INI-TR 1995-3.