Parallel I/O for Cluster Computing
by Christophe Cerin and Hai Jin (eds)
ISBN:1903996503
Kogan Page © 2004 (240 pages)

The aim of this publication is to present recent theoretical and practical advances that will assist potential new users and more experienced users in having a greater understanding of the subject of I/O management.

Table of Contents

Back Cover

Data sets in large applications such as databases and data-mining applications are often too large to fit into a computer's internal memory. The resulting I/O (input/output) communications between internal memory and the slower external medium (such as disks) can present a major performance bottleneck.

Among the possible computer architectural solutions to this problem are cluster computing platforms, which are collections of non dedicated systems (for example, PCs) based on low cost hardware, and these present themselves as affordable and suitable I/O systems.

The aim of this publication is to present recent theoretical and practical advances that will assist potential new users and more experienced users in having a greater understanding of the subject of I/O management. The first part of the book introduces selected topics and a number of effective solutions in different subtopics of the field. The second part presents five research studies contributed by specialist researchers in the field, located mainly in Europe and the USA.

About the Editors

Christophe Cérin is Associate Professor of Computer Science at the University of Picardie Jules Verne, France. He is a member of LaRIA (Laboratoire de Recherche en Informatique d`Amiens) and works with a team on parallel I/O systems. Dr Cérin is involved with different research programs including CGP2P, devoted to grid computing, and GRAPPE, devoted to cluster computing. He has published over 25 research papers and continues research work covering parallel I/O, experimental parallel algorithms for heterogeneous systems, memory hierarchy problems, and storage challenges for grid systems.

Hai Jin is Professor of Computer Science and Engineering at the Huazhong University of Science and Technology (HUST) in China. Dr Jin is a member of IEEE and ACM, and he is also the executive member and region coordinator of the IEEE Task Force on Cluster Computing (TFCC). He has served as program committee for more than 30 international conferences/workshops. The co-author of four books, Dr Jin has published over 50 research papers and has ongoing research interests in parallel I/O, high performance storage systems, cluster computing and grid computing, network security, and fault tolerance.

Parallel I/O for Cluster Computing

Christophe Cérin

Hai Jin

KOGAN
PAGE
SCIENCE

First published in Great Britain and the United States in 2004 by Kogan Page Science, an imprint of Kogan Page Limited

120 Pentonville Road
London N1 9JN
UK
http://www.koganpagescience.com

22883 Quicksilver Drive
Sterling VA 20166-2012
USA

1-9039-9650-3

British Library Cataloguing-in-Publication Data

A CIP record for this book is available from the British Library.

Library of Congress Cataloging-in-Publication Data

Parallel I/O for cluster computing / edited by Christophe Cérin and Hai Jin.
p. cm. - (Innovative technology series. Information systems and networks)
Includes bibliographical references and index.
ISBN 1-903996-50-3
1. Parallel processing (Electronic computers) 2. Electronic data processing-Distributed processing 3. Beowulf clusters (Computer systems) 4. Computer input-output equipment. I. Cérin, Christophe. II. Jin, Hai. III. Series.

QA76.58.P37635 2003
004'.35-dc21

2003054548

About the Authors

CHRISTOPHE CÉRIN is Associate Professor of Computer Science at the University of Picardie Jules Verne, France. He is a member of LaRIA (Laboratoire de Recherche en Informatique d'Amiens) and works with a team on parallel I/O systems. Dr Cérin is involved with different research programs including CGP2P, devoted to grid computing, and GRAPPE, devoted to cluster computing. He has published over 25 research papers and continues his research work which covers parallel I/O, experimental parallel algorithms for heterogeneous systems, memory hierarchy problems, and storage challenges for grid systems.

HAI JIN is Professor of Computer Science and Engineering at the Huazhong University of Science and Technology (HUST) in Chine. Dr Jin is a member of IEEE and ACM, and he is also executive member and region coordinator of the IEEE Task Force on Cluster Computing (TFCC). He has served on the programme committees of more than 30 international conferences/workshops. The co-author of four books, Dr Jin has published over 50 research papers and has ongoing research interests in parallel I/O, high performance storage systems, cluster computing and grid computing, network security, and fault

tolerance.

Foreword

While applications for mass markets (Internet) and also niche markets (scientific computing) are almost entirely data driven by nature, the ultimate performance of the machine will depend heavily on the quality of input/output (I/O) operations. Yet, all too often researchers in the parallel computing field have relegated the issue of I/O to the realm of assumption ("let us assume that the data has been pre-loaded in the various processors," or "suppose that data can be sent to the processors in linear time," or a myriad of other false premises). Instead, they have concentrated on the easier (which is not to say easy) problem of executing parallel complex applications and have allowed themselves to be driven by the simple law of computing which predicts the doubling of transistors on a chip every 18 months or so (Moore's law). However, at the same time, real disk storage densities have progressed at a rate of 60 to 80% per year but disk access time improvement has been less than 10% per year.

This has proven to be a pitfall. For one thing, Amdahl's law has warned us for a long time that it is the sequential portion of an application which dominates its performance and that further attempts at speeding up the parallel portion would be futile. Better to spend more design time reducing the amount of sequentiality than increasing parallelism or else risk investing too much effort in a cost ineffective approach! As it turns out, poor planning may easily turn I/O operations into a perfectly sequential stream of instructions which will bog down the most massively powerful parallel machines. To deny the existence of the I/O problem has all too often been a fundamental error.

To be sure, parallel computing had to undergo this first stage: it is hardly imaginable to build parallel machines without first making sure that they could be programmed efficiently. The fallacy was for us to imagine implementing easily shared disk storage to duplicate data on sites close to data requests instead of increasing local disk storage. Then, we would need also to increase the speed of the CPU to balance the increase in the amount of available data. Instead, parallel I/O corresponds to this idea of sharing affordable (and small) disks and to keep them as busy as possible to get the best throughput of data.

Consequently, it is quite heartening to find this book which reflects the new challenges and opportunities of parallel I/Os. The authors have contributed in an original manner to our recognition of the I/O issue as a whole and have endeavored to take you, the reader, on a journey of understanding through this difficult and sometimes ignored topic.

This book is organized into two parts: an introduction to selected topics and a selection of effective solutions in different subtopics of the field.

Part one introduces some motivating I/O problems both for the sequential case and the parallel one. Sequential I/O means in general that I/O requests reach the processor, driver, interface one after the other, while parallel I/O means that there is potentially more than one concurrent I/O operation at any given time. The chapters which make up part one include a study of optimizing the implementation of distant reading/writing (from disk to disk). Then the product of matrices is introduced to illustrate how MPI-2 captures different I/O abstractions. Sequential sorting on disks is presented to emphasize the differences between in-core algorithms and out-of-core algorithms while special optimization techniques (polyphase mergesort) are described. Further, the authors have not ignored the importance of data structures and special attention has been given to this problem (B-trees, special purpose trees, etc.). The authors have recognized the importance of heterogeneity for the future of computing and devote most of Chapter 2 to parallel sorting on heterogeneous clusters.

The second part of the book introduces five research studies: a study of parallel I/O under a parallel file system (Chapter 3), a presentation of benchmarking I/O (Chapter 4), parallel join algorithms on clusters (Chapter 5), a study about scheduling I/Os (Chapter 6) and the design and implementation of a large cache for software RAID (Chapter 7).

It has been an honor to be asked to write this foreword. Indeed, I have enjoyed reading this manuscript. The wide spectrum of topics covered is a testimony to the vitality of the field and I believe that the contributions of this book will be invaluable to students and practitioners in the field.

Jean-Luc Gaudiot
Professor
University of Irvine, California, USA

Introduction

Data sets in large applications such as databases and data-mining applications are often too large to fit entirely inside the computer's internal memories. The resulting input/output communications (or I/O) between internal memory and slower external medium (such as disks) can be a major performance bottleneck: the current ratio between memory access time and disk access time is still about 10. In this context, exploiting locality is one of the very important challenges in the field.

Among the computer architectural solutions proposed to deal with the challenges, we have today the very effective "Cluster Computing Platforms" [BUY 99a, BUY 99b] which are collections of non dedicated systems (for instance PCs) based on low cost hardwares. Thus clusters present themselves as affordable and suitable I/O systems.

Cluster architecture

At the end of the 90s, *networks of workstations* or *clusters* appeared as a variety of distributed architecture sharing "more than nothing" as one says frequently. The "sharing" can consist of sharing the configuration of machines or in a more technical way, the space of addressing files or memory, to globalize the physical resources of the various machines.

This architecture is a reasonable alternative [BUY 99a, BUY 99b] to dedicated parallel machines. It is now noticed that the main factors that led to this acceptance are connected to economic factors: the ratio between cost of the machine and the performance are very advantageous when one can count on mass product component.

A network of workstations consists of:

- standard components (microprocessors, DRAM, disks...) such as one buys in big distributors;
- general and affordable motherboards (one finds also bi-processor cards for as little as 150 euros);
- networks of interconnection based on a tried standard: Fast-Ethernet (100Mbits/s) and Gigabit-Ethernet (1Gbits/s) or dedicated networks such as Myrinet who offer communications of more of 1Gbits/s measured with a latency of the order of 10μs. These performances are of the same order as those of the networks of parallel computers as the IBM SP... while the cost of a Myrinet card is about 1000$USD (only);
- parallel programming libraries, generally by sending messages with MPI or BspLib [MIL 94] or BUP7 [BON 99] which implement the BSP model [VAL 90].

Besides the advantageous economic aspect, the networks of workstations allow *scaling*: the extension of a system is very easily realized and very often, at a very reasonable cost. Indeed, in architecture of client/server type, when the number of clients increases, the server is the bottleneck because it is by definition the only one to return a service. In shared memory architecture, the traffic on the bus becomes a bottleneck, generally from a dozen nodes. In architecture of cluster type, the data are distributed on all the machines and potentially reduce problems of bottleneck.

Managing Inputs and Outputs (I/O): a key challenge

Storage systems represent a vital and growing market and deal with cached, historical, multimedia and scientific data. As the ubiquity of clusters has grown in the past decade by the use of appropriate System Area Networks (SAN), we still have to prove that the "cluster paradigm" is a serious candidate to handle large data sets. One way to achieve good performance, is to allow I/O operations to perform more and more in parallel since a typical disk drive is 10^5 times slower in performing a random access than is achieved in the main memory of a computer. The factor can be 10^6 and even more if we consider the access time of an internal CPU register of an 1Gz PC.

This book is devoted to the recent evolution of Parallel I/O [JIN 01, KOR 97, MAY 00] in the context of cluster computing field. It presents "state of the art" contributions on the subject of I/O management. Thus, the aim of the book is also in the presentation of recent, practical and theoretical advances that could help potential new users in the field. The book attempts to cover the main topics:

- File Systems and Parallel I/O for Clusters;
- Data Distribution and Load Balancing in the Presence of I/O Operations;
- Novel Hardware and Software I/O Architectures;
- Parallel Disk Models and Algorithms;
- Parallel I/O Support for Databases;
- I/O Performance Analysis: Resources and Tools for Benchmarking;
- Drivers and Application Programming Interfaces;
- Advances in Storage Technology;
- Tools for Operating and Managing I/O Operations;
- Compilers Techniques for High Performance I/O Operations;
- Language and Runtime Libraries;
- Network Attached Storage and Storage Area Network;
- Standards and Industrial Experiences with Massive Data Sets.

The book covers in depth the first eight topics. It is divided into two parts. In the first part we introduce all the necessary vocabulary and we exemplify some particular problems. Our goal is both to motivate the reader and to survey some widely used and general problem/solutions. The part could be used to introduce the field to students: it is educationally oriented. It covers in the first chapter some solutions to accelerate disk transfers by working at the driver level; then we focus on how MPI-IO allows one to specify I/O operations and we introduce algorithmic issues for sequential sorting. We also focus on tree data structures and prefetching techniques. Chapter 1 ends by considering disk scheduling algorithms. In chapter 2, we develop recent advances in parallel out-of-core (external) sorting in the case of an heterogeneous cluster.

In the second part of the book, we present recent approaches and solutions that researchers in the field have contributed in the last two years.

The book is not really a textbook in the sense that it is not organized along problem statements, questions, pauses, exercices, problems and research. We want this book to help the reader to understand better the fundamental questions and effective answers underlying the I/O problems. We introduce in the first part some fundamental notions motivated by examples, and then in the second part, we present recent research advances that use some prerequisites presented in the first part. We hope that this book will help readers to produce efficient I/O programs in the future and also will help educators to find sources of exercises.

We guess that the reader is familar with the PC hardware as it is taught at undergraduate course level [CLE 00, DAV 98, HEN 02] and with operating systems [TAN 01, STA 01, NUT 01] and has some background about the following notions in order to facilitate the reading of the first part:

- Parallel programming with MPI;
- Parallel algorithms [AKL 97, JÁJ 92] (complexity notion of (parallel) algorithms);
- Unix file system;
- Benchmarking;

- Sort algorithms and join operation in databases [ULL 02];
- Cache and RAID systems.

The work would not have accomplished without the key efforts of the editorial board as listed below. We would also address a special thank to Rajkumar Buyya who has been one of the main supporter of the book idea. His strong involvement in the IEEE Task Force on Cluster Computing has been a model for our motivation.

Guest Editors:

Christophe Cérin
Université de Picardie Jules Verne
LaRIA, bat Curi,
5, rue du moulin neuf
80000 AMIENS - France
Email: c.cerin@computer.org
URL: http://www.u-picardie.fr/~cerin

Hai Jin
Internet and Cluster Computing Center
Vice-Dean, College of Computer
Huazhong University of Science
and Technology
Wuhan, Hubei, 430074 - China
Email: hjin@hust.edu.cn
http://www.hust.edu.cn/hjin/

Editorial Board:

Claude Timsit,
(PRiSM/Versailles & SUPELEC),
Email: Claude.Timsit@supelec.fr

Gil Utard,
(ENS/Lyon),
Email: Gil.Utard@ens-lyon.fr

Jean-Luc Gaudiot,
(UCI/Los Angeles),
Email: gaudiot@uci.edu

Erich Schikuta,
(Vienna, Austria),
Email: schiki@ifs.univie.ac.at

Franck Cappello,
(LRI/Orsay),
Email: fci@lri.fr

Rajkumar Buyya,
(Monash University),
Email: rajkumar@csse.monash.edu.au

Toni Cortes,
Barcelona School of Informatics
Universitat Politecnica de Catalunya
Email: toni@fib.upc.es

Rajeev Thakur
Mathematics and Computer
Science Division
Argonne National Laboratory
Email: thakur@mcs.anl.gov

Yiming Hu
Dept. of Electrical & Computer
Engineering & Computer Science
University of Cincinnati
Email: yhu@ececs.uc.edu

Michel Tréhel
(Université de Franche-Comté)
Email: trehel@lifc.univ-fcomte.fr

Evgenia Smirni
Department of Computer Science
College of William and Mary,
Williamsburg, VA 23187-8795
Email: esmirni@cs.wm.edu

Peter Brezany
Institute for Software Technology
and Parallel Systems
University of Vienna, Austria
Email: brezany@par.univie.ac.at

PREV < Day Day Up > NEXT

General interest bibliography

[AKL 97] Akl S., *Parallel Computation, Models and Methods*, Prentice Hall, 1997.

[BON 99] Bonorden O., Juurlink B., von Otte I., Rieping I., *"The Paderborn University BSP (PUB) Library - Design, Implementation and Performance"*, *13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing, 12 - 16 April, 1999, San Juan, Puerto Rico, available electronically through IEEE Computer Society*, 1999.

[BUY 99a] Buyya R., *High Performance Cluster Computing, Volume 1: Architectures and Systems*, P T R Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1999.

[BUY 99b] Buyya R., *High Performance Cluster Computing, Volume 2: Programming and Applications*, P T R Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1999.

[CLE 00] Clements A., *The Principles of Computer Hardware (third Edition)*, Oxford University Press, 2000.

[DAV 98] David Culler J. S., Gupta A., *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann Publishers, 1998.

[HEN 02] Hennessy J., Patterson D., *Computer Architecture, A Quantitative Approach (third edition)*, Morgan Kauffmann, 2002.

[JÁJ 92] JÁJÁ J., *Introduction to Parallel Algorithms*, Addison Wesley, 1992.

[JIN 01] Jin H., Cortes T., Buyya R., Eds., *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, IEEE Computer Society Press and Wiley, New York, NY, 2001.

[KOR 97] Korfhage R. R., *Information Storage and Retrieval*, John Wiley & Sons, 1997.

[MAY 00] May J. M., *Parallel I/O for High Performance Computing*, Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 2000.

[MIL 94] Miller R., Reed J., The Oxford BSP Library: User's Guide., Report, Oxford University Computing Laboratory, 1994.

[NUT 01] Nutt G., *Operating Systems: A Modern Perspective, Lab Update*, Addison Wesley, 2001.

[STA 01] Stallings W., *Operating Systems: Internals and Design Principles, 4/e*, Prentice Hall, 2001.

[TAN 01] Tanenbaum A. S., *Modern Operating Systems, 2/e*, Prentice Hall, 2001.

[ULL 02] Ullman J. D., Widom J. D., *First Course in Database Systems, A, 2/e*, Prentice Hall, 2002.

[VAL 90] Valiant L., *"A Bridging Model for Parallel Computation"*. *Communications of the ACM.*, num. 33, p. 103-111, August 1990.

Part One: Cluster Computing and I/Os

Chapter List

Chapter 1: Motivating I/O Problems and their Solutions

Christophe Cérin, Université de Picardie Jules Verne, LaRIA, 5 rue du moulin neuf, 80000 Amiens, France
Hai Jin, Huazhong University of Science and Technology, Wuhan, 430074, China

Overview

In this chapter we present an introduction to the high performance storage field when the demands of I/Os take place either in sequential or in parallel, which is the main concern of the book. We consider this introduction as a point of entry to the disciplinary field. We do not claim to raise an exhaustive list of all the problems connected to disks: we made a personal choice. For example we assume that the parallel underlying system is a cluster of PCs and that it is totally dedicated. So, we shall not evoke the systems of disks such as they exist in industrial environments for example. The objective of this part is to show various techniques to obtain performances when one uses disks. We assert that efficient techniques allows one very often to obtain significant improvements in the computer's overall performance.

We have selected topics on the methodological plan (file system design), on low level affairs (conception of intelligent drivers), on algorithms (disk scheduling), on data structures (trees) and compiler optimizations (prefetching techniques).

Let us remind ourselves that we have to use external algorithms/solutions either because the problem size does not fit in main memory or because one wishes deliberately to limit the allowance of RAM by the applications: in a context of non-dedicated cluster, one can imagine that some processes are not privileged and can use only little internal memory (some percentage of the whole RAM).

1.1 Technological facts

Software applications for Internet such as digital bookshops, virtual laboratories, video on demand, electronic business, WEB services, collaborative systems demand mastery of data storage, I/O operations, balancing of the loads of the systems, protection of data and long distance communication delays.

It is known well that the power of processors doubles every 18 months. It is Moore's law. Concerning disks, although the density of the substrate increased in the order of 60 - 80% a year, the data access time, which depends largely on mechanical factors, are known only to increase by a factor of 10% a year. The immediate consequence is that the system of disks became a bottleneck. More than ever, it is important to study techniques to by-pass this difficulty. The first technique consists in multiplying disks and in ordering the computations so that they reach, in a concurrent way, the data. In a ideal way this solution allows one potentially to store a big mass of data while minimizing the losses of performance.

Obviously, to get performance, there are other material or software techniques such as the use of caches, but we shall not approach them here. The chapters in part 2 of the book will consider this later.

1.2 Working at the driver level

A part of the work of Olivier Cozette[1] aims to have no cache but to realize efficient readings and successful distant writings. This work [COZ 02] is carried out at the level of the network/disk drivers. The discussions in this section illustrate perfectly the benefits that one can obtain by organizing the data and by controlling the manner in which we accomplish disk transfers. The objective pursued by Olivier Cozette is to offer a parallel file system (offering a single address space) with I/O performance based on direct readings and writings to disks.

The target architecture in this section is a dedicated cluster, of which nodes are typically motherboards of PC. The architecture is shown in Figure 1.1 and the network of communication in the experiments is Myrinet. The disks are SCSI disks.

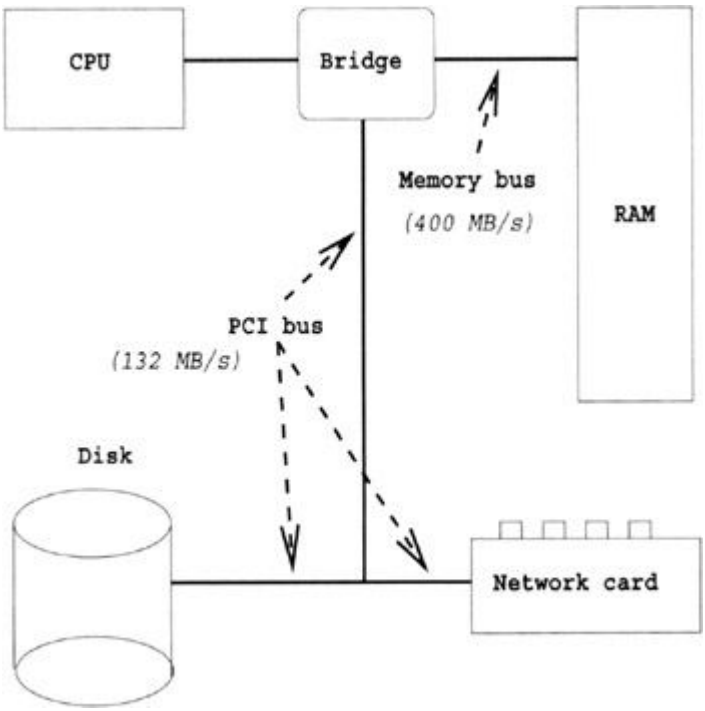


Figure 1.1: Architecture of a PC card

One of the conditions to obtain performances is to seek, at the time of a reading (writing), the least architectural component possible. For example, how one can make disk to disk transfers without mobilizing cycles of the CPU? When it is not carefully accomplished, a disk to disk transfer requires a copy of data to the disk and towards the RAM via the PCI bus and after that via the memory bus, then a copy from the RAM towards the network card. Then finally there is a network transfer which generates a highly-rated traffic, from the receiver side. See Figure 1.2 to follow the three movements of data.

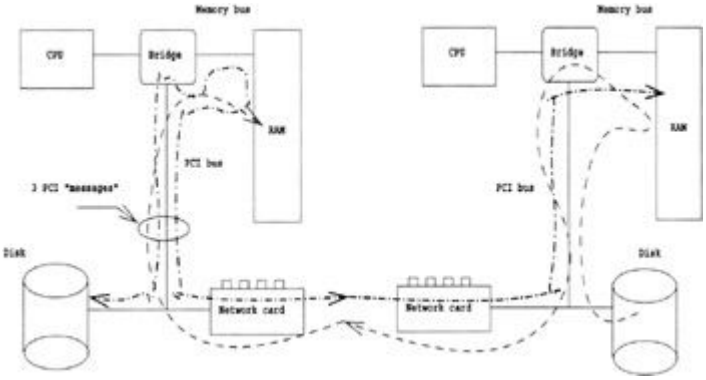


Figure 1.2: Activity of buses with two concurrent readings

Let us imagine that two sites ask simultaneously to read data on the distant disk from the other one. After what we have just said, it is easy to verify that the maximum throughput in reading for every disk is the throughput of the bus PCI (the slowest) divided by three (see Figure 1.2). Moreover, the CPU is mobilized because the reading is made via a system call.

In re-writing the SCSI driver of the Linux operating system, one can avoid one copy towards the memory as is presented in Figure 1.3. This solution does not require either a plug at the user level (the programmer does not need to re-write its code) or the addition of a particular file system.

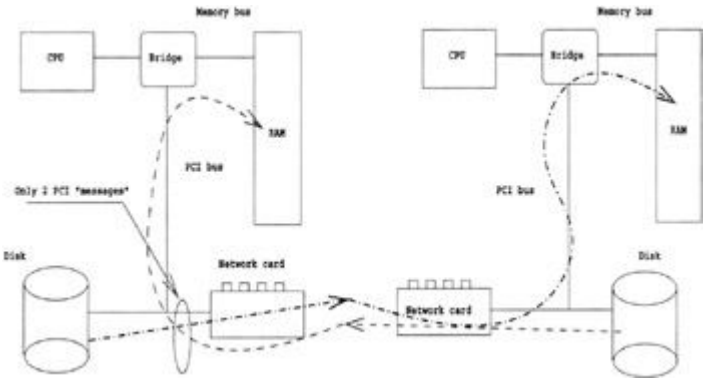


Figure 1.3: Optimized concurrent readings

The result of experiment is presented in Figure 1.4. One shows that a direct access to the disk improves the throughput of reading of disks by a factor of about 25%. Because the memory bus is less sought, it is normal also to obtain a better throughput of the memory for the associated application which is a scan[2].

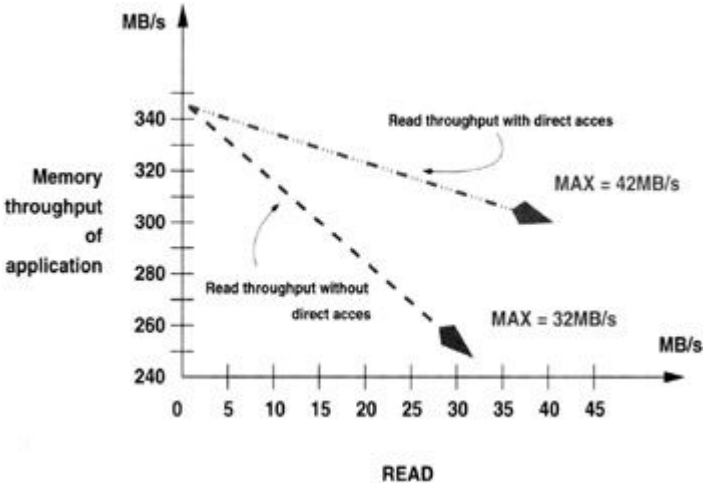


Figure 1.4: Experimentation: direct access versus normal access

The values of 42MB/s and 32MB/s correspond each to those one can obtain better! In fact, on the curves of Figure 1.4, there is yet another parameter which corresponds roughly to the time which passes between two reading demands: the throughput of reading is null for points on the vertical axis because in this case we are only waiting. Let us notice finally that the technique used here aims not to build a cache in order to amortize transfer costs. We notice also that in mastering the I/O operations we also reduce the memory (to/from the RAM) traffic.

[1]Olivier Cozette (Amiens - France) can be reach at cozettei@laria.u-picardie.fr

[2]A scan is for instance the computation of a sum of entries in an array

1.3 Parallel programming and I/Os

The matrix multiplication on disks is the application presented in this paragraph. The problem illustrates the way of organizing the distribution of data in order to minimize the access costs of data on disks. In fact, we show how MPI allows coding of the data distribution. In other words, it is a question of reporting the view offered to the programmer of parallel applications to specify the distribution and the scheduling of accesses on disks.

We introduce now different MPI-I/O implementations for the out-of-core product of matrices. The key idea is to illustrate, through examples, how the programmer puts in correspondence the physical organization of data and the logical organization induced by the algorithm.

Basically, a MPI-IO file is an ordered collection of typed data. Each process of a parallel MPI-IO program may define a particular *view* of a file and that view represents an ordered subset of a file. A view is defined by a MPI *basic* or *derived* data type (a derived datatype is any data type that is not predefined) and by a displacement (it can be used to skip headers). Separate views, each using a different displacement and filetype, can be used to access each segment of a file, thus the views of different processes may overlap and the overlapped data are then shared between processes. The MPI-IO standard introduces also specific operations to tackle the consistency problem of overlapping file views but they are not used in our case study.

As pointed out by Thakur in [GRO 99b], a programmer of out-of-core applications can use neither derived data type nor collective (read/write) operations to access data or can use only derived data type or he employs only collective operations or, at least, he uses a combination of derived data type and collective operations. The text below exemplifies such usages.

1.3.1 The product of matrices on disk

We do remind briefly the sequential (in-core) algorithm of the matrix product of two $n \times n$ matrices with the following pseudo-code:

```
for(i=0;i<n;i++)
    for(j=0;j<n;j++){
       C[i,j] = 0;      /* A and B are the given matrices */
       for(k=0;k<n;k++) /* of size n * n                  */
          C[i,j] = C[i,j] + A[i,k] * B[k,j];
    }
/* At the end, matrix C contains the product */
```

The number of products accomplished by the code is $\mathcal{O}(n^3)$. A naive out-of-core algorithm can follow the previous code by replacing the (in-core) read instructions by file reading instructions. It is not really a good idea but it works! We will explain later why such translation strategy is not good practice, for instance in the case of a tree traversal for a tree stored on disks.

Let us suppose that the required distribution of the data is that of Figure 1.5.

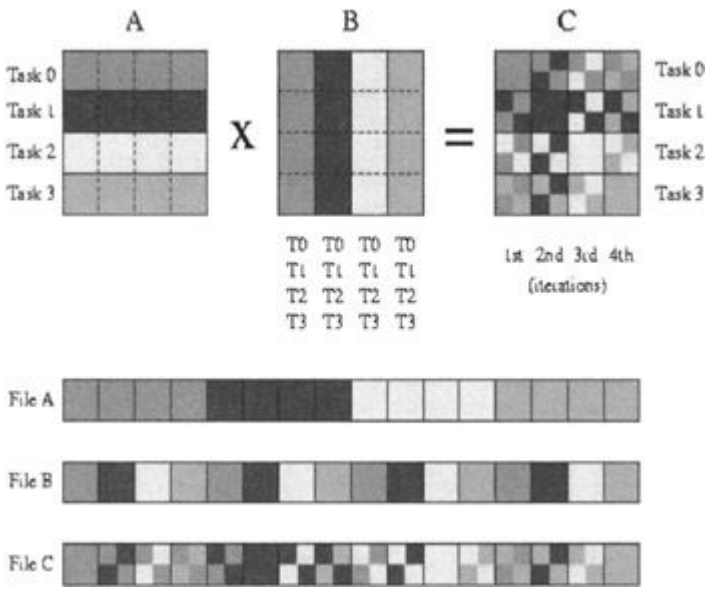


Figure 1.5: Matrix product

1.3.2 MPI-2 primitives of data distribution management

The Message-Passing Interface (MPI) specification [GRO 99a, FOR 94] is widely used for solving significant problems with parallel computers. There exists many implementations of MPI on computer platforms and they are often optimized for each platform. The documentation about I/Os can be found on MPI forum web site[3].

One of the objectives pursued in MPI-2 for the management of I/O operations aims to define a universal representation of the data on disks which transforms itself from the representation to concrete data (this is a question of implementation and of portability). In addition, further efficiency can be gained via the control over physical file layout on storage devices (disks).

In the remainder of the section we follow the presentation of Jean-Pierre Prost as can be found on the web at:

`http://www.pdc.kth.se/training/Talks/SMP/maui98/jpprost/s1d001.htm`

The abstract type "file" declines then as it is presented in Figure 1.6. The programmer can code that data are contiguous on disk (*contiguous type*), or spaced in a regular way (*vector*), or accessed by an index (*indexed type*) or structured: different data types can be recorded (*struct mode*).

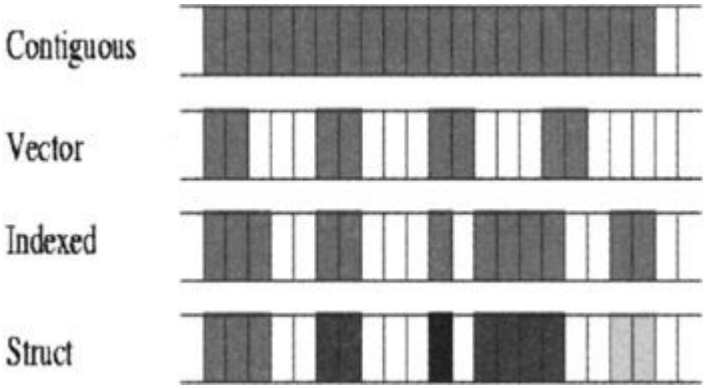


Figure 1.6: The abstract type "file"

It is necessary to note that constructors of the abstract type can be used recursively. The user can also specify a movement with regard to the beginning of the file (that allows one to pass on a header for example) as well as information about access to the data (sequential, random). The user shall also specify if the data representation allows some degree of "inter-operablility". The major mode file that the programmer should use are:

native: Data in this representation is stored in a file exactly as is in memory. A file cannot be used on a system which has not created it;

internal: The data representation can be used for I/O operations in a homogeneous or heterogeneous environment. The file is useful on any system supporting the implementation which created it.

1.3.3 A framework for a possible implementation of matrix product

The algorithmic scheme to deploy is listed in Appendix 1. A final implementation can be found on:

```
/*
 * MPI-2: Out-of-Core Matrix Product - Complete C Code
 * http://www.pdc.kth.se:81/training/Talks/SMP/maui98/jpprost/ex2_c_comp.htm
 *
 * Author: Jean-Pierre Prost
 * Email: jpprost@us.ibm.com
 * Home Page: http://www.research.ibm.com/people/p/prost
 *
*/
```

As an exercise, the reader is invited, after downloading the code, to run it and to observe which part of the matrix is local to each processor (disk) and to justify, in the code, the description of data distribution.

In Appendix 1, the reader will find a code for the product:

$$C = A * B = \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{24} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} * \begin{bmatrix} B_{11} & B_{12} & B_{13} & B_{14} \\ B_{21} & B_{22} & B_{23} & B_{24} \\ B_{31} & B_{32} & B_{33} & B_{24} \\ B_{41} & B_{42} & B_{43} & B_{44} \end{bmatrix}$$

according to the physical data representation for the file containing matrix A:

$$A_{11}, A_{12}, A_{13}, A_{14}, A_{21}, A_{22}, A_{23}, A_{24}, A_{31}, A_{32}, A_{33}, A_{24}, A_{41}, A_{42}, A_{43}, A_{44}$$

and for the file containing matrix B:

$B_{11}, B_{21}, B_{31}, B_{41}, B_{12}, B_{22}, B_{32}, B_{42}, B_{13}, B_{23}, B_{33}, B_{43}, B_{14}, B_{24}, B_{24}, B_{44}$

The reader is invited to explain the difference in coding between the code listed in Appendix 1 and the code of Jean-Pierre Prost.

Our main concern in this section and in Appendix 1 was to illustrate the step of coding in MPI-I/O and in particular how to use the abstract data type specifications.

1.3.4 Strassen algorithm

In 1969, Strassen [STR 69] proposed a faster algorithm than the previous $\mathcal{O}(n^3)$ algorithm in order to multiply two matrices of size $n \times n$. The method is based on the substitution of multiply by addition operations because additions run more quickly than multiply operations. The complexity of the method is $\mathcal{O}(n^{\log 7})$ $n^{2.81}$.

The Strassen method[4] is based on the following decomposition for the product:

$$C = A * B = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} * \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$= \begin{bmatrix} M_0 + M_1 + M_2 - M_3 & M_3 + M_5 \\ M_2 + M_4 & M_0 - M_4 + M5 + M_6 \end{bmatrix}$$

with

$$M_0 = [A_{11} + A_{22}] * [B_{11} + B_{22}]$$

$$M_1 = [A_{12} - A_{22}] * [B_{21} + B_{22}]$$

$$M_2 = A_{22} * [B_{21} - B_{11}]$$

$$M_3 = [A_{11} + A_{12}] * B_{22}$$

$$M_4 = [A_{21} + A_{22}] * B_{11}$$

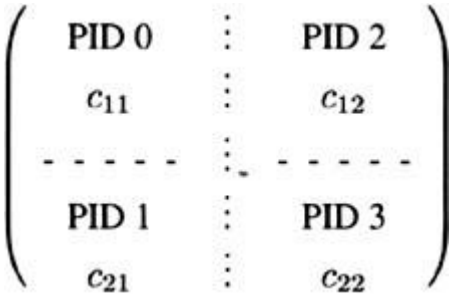
$$M_5 = A_{11} * [B_{12} - B_{22}]$$

$$M_6 = [A_{21} - A_{11}] * [B_{11} + B_{12}]$$

Note that all matrices in the previous formula are of size $\frac{n}{2} \times \frac{n}{2}$. The skeleton of a sequential in-core solution of the Strassen method is:

```
void multiply (int n, matrix a, matrix b, matrix c, matrix d)
{
    if (n <= BREAK). {
        double sum, **p = a->d, **q = b->d, **r = c->d;
        int i, j, k;
        /* Use the classical 0(n^3) algorithm */
        for (i = 0; i < n; i++) {
            for (j = 0; j < n; j++) {
                for (sum = 0., k = 0; k < n; k++)
                    sum += p[i] [k] * q[k] [j];
                r[i] [j] = sum;
            }
        }
    } else {
        n /= 2;
        sub(n, a12, a22, d11);
        add(n, b21, b22, d12;
        multiply(n, d11, d12, c11, d21);
        ..............................
        /* compute recursively other submatrices */
    }
```

A skeleton of a MPI-2 code is provided in Appendix 1. The code does not match exactly the previous recursive decomposition. We also assume that all the matrices are copied on the local disks of the processors in the cluster. Note that in the proposed implementation we assume that processors are in charge with the data according to the following decomposition:



For instance, processor 0 is in charge with the computation of the upper left corner of matrices c, namely c_{11}. The definition of a_{11}, b_{11} submatrices, in the code, is made by two `MPI_Type_create_subarray` MPI-I/O instructions. We provide also one incore solution for a "Strassen like" algorithm. The sequential implementation involves no level of recursion. It is more simple to implement it. It does not match the bound of $n^{2.81}$ operations: it involves only submatrices of sizes $n/2$. It remains to parallelize it!

The main interest of the code fragment, in terms of I/O operations and comparing it with the two previous ones is to show another way to code an external data distribution that is implied by the algorithm. The reader is invited to point out the different steps of the computation done in the code (Strassen implementation, see Appendix 1) in order to master MPI-2 and I/O operations and to answer the following questions:

Classical method for the product:

1. What is the space complexity (memory used) by the two codes (Prost code available on his web site and or code (see Appendix 1))? Give an estimate. Compare the two complexities?

2. For large n, do you think that the code of Appendix 1 will run faster than the code of Prost? In the exercise, be careful of the parallel actions!

Concerning Strassen algorithm:

1. Find the dependences beetween instructions in the sequential `multiply_strassen` procedure (see Appendix 1). For instance, in the following code fragment:

```
/* I1 */      add_in_core(n, a11, a22, d1);
/* I2 */      add_in_core(n, b11, b22, d2);
/* I3 */      multiply_in_core(n, d1, d2, M0);
```

   we have that instruction $I3$ uses $d1$, $d2$ produced by instruction $I1$, $I2$ respectively. So, instruction $I1$, $I2$ can be executed in parallel and then we are allowed to execute $I3$ only after the completion of $I1$, $I2$.

2. After the data dependence isolation, derive a parallel code based on the code fragment listed in Appendix 1. The parallel code will be such that processor 0 will compute c_{11}, processor 1 will compute c_{21}. processor 2 will compute $c12$ and processor 3 will compute $c22$. Each processor has a local copy of the two given matrices. Add enough `MPI_Type_create_subarray` and `MPI_File_set_view` instructions in order to facilitate the programming. Please refer to

   http://www.mpi-forum.org/docs/mpi-20-html/node171.htm#Node171

   for the complete documentation about MPI-2.

3. Modify the implementation of Strassen algorithm (see Appendix 1 and the previous question) in order to have a minimum number of sub-matrices loaded at a time in the main memory of a processor. In another words, use disks as much as possible.

[3] See: http://www.mpi-forum.org/docs/docs.html

[4] http://www.supercomp.org/sc96/proceedings/SC96PROC/JACOBSON/INDEX.HTM

1.4 Sequential sorting with disks

In this section, we concentrate on the example of sequential sorting. What differentiates it with regard to the previous problem (product of matrices) comes from the nature of the memory accesses which are unpredictable. So, a good initial distribution of data cannot be envisaged at first to pull the performances upward.

We present sequential versions of external sorting: two-way merge sort, balanced two-way merge sort, balanced k-way merge sort and the polyphase merge sort. These algorithms are well known since the trilogy by Knuth published in 1973 and republished in [KNU 98]. Techniques used for external sorting are quiet different from techniques used for in-core sorting. Moreover, we will use a sequential external sorting algorithm as a basic block of the parallel external algorithm we will introduce in a forthcoming chapter.

1.4.1 Two-way merge sort

In a general way, a sorting algorithm works in two stages:

1. The records of the file to be sorted are divided into several groups (called "runs" in the literature), every group fits in the main memory. An internal sorting algorithm is applied to every group and the sorted groups are distributed in two files;
2. Alternately, every sorted group is merged to form a bigger group. The result is redistributed in both initial files; one continues until we form more than a single group of data of which are sorted.

Example:

Let us suppose that we have a file with the following keys:

```
50 110 95 10 100 36 153 40 120 60 70 130 22 140 80
```

Suppose that the size of a group is 3. We can divide the file as follows:
1(50, 110, 95,) 2(10,100, 36), 3(153,40,120), 4(60,70,130), 5(22,140,80).

These five groups are brought in main memory in order to be sorted. Then, they are distributed in two files in the following way (alternate distribution that favors certain balancing of loads):

```
File 1 contains the groups 1 ( 50,110,95 ), 3 ( 153,40,120 ),
and 5 ( 22,140,80 ).

File 2 contains the groups 2 ( 10,100,36 ) and 4 ( 60,70,130 )
```

Now we merge both files (groups) in a third file, a group at a time: the group 1 of the file 1 is merged with the group 2 of the file 2 to produce the first group of file 3, etc. One redistributes the groups of file 3 in both files 1 and file 2. Now, one begins again the fusion and not stage 1 of the algorithm because the groups are sorted... until we make an unique group.

The number of stages of the algorithm is $[\log_2 N]$ where N is the number of initial groups. A logarithmic number of stages is considered to provide a good improvement compared with a linear (which means N passes) number of stages of the following and naive algorithm:

```
Step 0: mark all entries as "not visited";
Step 1: find the smallest element, store it in the output file,
        mark it as "visited" in the input file;
Step 2: return to step 1 until the input file is not empty
```

1.4.2 Balanced two-way merge sort

This is an improvement of the previous algorithm by considering a bigger number of used files. The "balanced two-way merge" algorithm merges two files and arranges the merged groups in two output files and in an alternative way. This eliminates the need to redistribute the groups.

The algorithm merges the groups of file 1 and file 2, and arranges the merged groups in file 3 and file 4. Then the algorithm merges the groups of files 3 and 4 and arranges the results in files 1 and 2. The algorithm ends when one obtains a single group.

The complexity of balanced two-way merge sort, in terms of passes over the files, is the same as the previous algorithm.

1.4.3 Balanced k-way merge sort

It is also an improvement on the previous algorithm for sorting. One notices indeed that by increasing the number of input and output files, the number of groups in every file decreases because there are fewer fusions.

However, the phase of fusion is more complicated when one has *k* input files because one needs to distribute the sorted groups in *k* output files.

Let us resume the numeric example presented first. "Balanced 3-way Merge Sort" works as follows for the phase of fusion: group 1 of file 1, group 2 of file 2 and group 3 of file 3 are merged in file 4 to produce a group of sorted data. The group 4 of file 1 and group 5 of file 2 are merged in file 5 (File 6 is not used here because there are not enough data). Files 4 and 5 become the input files and both groups are merged to give the result. Generally speaking, the number of passes of the algorithm then becomes $[\log_k N]$.

1.4.4 Polyphase merge sort

This algorithm is an improvement of "k-way merge sort" face to face of the number of files used by the algorithm. In fact it uses an empty file in the following way. Let us resume the initial example (for the two-way merge sort). It involves 15 records. The beginning is the same: the group 1 of file 1 is merged with the group 2 of file 2 and produces the group 1 of file 3. Then, group 3 of file 1 is merged with group 4 of file 2 and produces the group 2 of file 3.

At this moment, file 2 becomes empty but file 1 has another group. And it is here that the novelty intervenes: the algorithm continues to merge file 1 and file 3 in putting the results in file 2. As soon as file 1 becomes empty, it becomes, in its turn, the new output file.

This process is repeated as often as required until we obtain a single group in one of the files. Note that the algorithm is effective only in the case when we have an uneven distribution at first, that is to say that we have not the same number of groups in files at the initial step of the algorithm.

It is by means of s special distribution named the *perfect Fibonacci distribution* that one manages to produce an uneven or unbalanced distribution. One can observe it in our previous example or the example that follows. Suppose that we begin with 8 records:

```
Original file:          we have 8 runs
Initial distribution:   5 and 3 runs
After phase 1:          3 and 2 runs
After phase 2:          2 and 1 runs
After phase 3:          1 and 1 runs
Finally:                1 run
```

All the numbers listed here (8, 5, 3, 2, 1) are Fibonacci numbers. Indeed if one has the possibility of using more intermediate files than 2 (for example in the case of the *k*-way merge sort), we then use the generalized definition of Fibonacci numbers:

Definition:

The Fibonacci numbers of order *p*, noted $F_n^{(p)}$ are defined as follows:

$$\begin{aligned} F_n^{(p)} &= 0 && \text{for } 0 \le n \le p-2 \\ F_n^{(p)} &= 1 && \text{for } n = p-1 \\ F_n^{(p)} &= F_{n-1}^{(p)} + F_{n-2}^{(p)} + \cdots + F_{n-p}^{(p)} && \text{for } n \ge p \end{aligned}$$

Example:

The Fibonacci numbers of order 3 are given by the following sequence (with the first one given by $F_0^{(3)}$):

0,0,1,1,2,4,7,13,24,44,...

Note that, by definition the Fibonacci series always begin with $p-1$ zero followed by one one. If the total number of records (runs) in a file is not a Fibonacci number, one adds markers to respect the property.

In general: at the beginning of a phase of fusion of a p-way merge sort, the *p* input files contain:

$$\underbrace{F_n^{(p)}}_{[1\text{ term}]}, \underbrace{F_n^{(p)} + F_{n-1}^{(p)}}_{[2\text{ terms}]}, \underbrace{F_n^{(p)} + F_{n-1}^{(p)} + F_{n-2}^{(p)}}_{[3\text{ terms}]}\}, \cdots, \underbrace{F_n^{(p)} + \cdots + F_{n-p+1}^{(p)}}_{[p\text{ terms}]}$$

groups, respectively. During this phase, $F_n^{(p)}$ groups will be merged leaving the number of groups following in every file to be:

$$0,\ \underbrace{F_{n-1}^{(p)}}_{[1\text{ term}]},\ \underbrace{F_{n-1}^{(p)}+F_{n-2}^{(p)}}_{[2\text{ terms}]},\ \cdots,\ \underbrace{F_{n-1}^{(p)}+\cdots+F_{n-p+1}^{(p)}}_{[p-1\text{ terms}]}$$

Furthermore, the output file will contain:

$$F_n^{(p)} = \underbrace{F_{n-1}^{(p)}+\cdots+F_{n-p}^{(p)}}_{[p\text{ terms}]}\ \text{groups}$$

As a consequence of the desired property, any polyphase merge sort algorithm has to begin with the construction of unbalanced runs, for instance with the following code:

```
See also: http://www.nist.gov/dads/HTML/polyphmrgsrt.html
and http://www.dcc.uchile.cl/~rbaeza/handbook/algs/4/444.sort.c.html
and also http://oopweb.com/Algorithms/Documents/Sman/VolumeFrames.html?/
        Algorithms/Documents/Sman/Volume/ExternalSorts_files/s_ext.htm

void PolyPhaseMergeSort(void) {
   /*
    * initialize merge files
    */
    initTmpFiles();
   /*
    * Function makeRuns calls readRec to read the next record.
    * Function readRec employs the replacement selection algorithm
    * (utilizing a binary tree) to fetch the next record,
    * and makeRuns distributes the records in a Fibonacci distribution.
    * If the number of runs is not a perfect Fibonacci number, dummy
    * runs are simulated at the beginning of each file.
    */
    makeRuns();
   /*
    * Function mergeSort is then called to do a polyphase
    * merge sort on the runs.
    */
    mergeSort();
   /*
    * cleanup files
    */
    termTmpFiles(0);
}
```

1.5 Tree data structures

The problem of high performance storage when the demands of I/O operations take place in parallel or sequentially is approached now with the *representation problem* in mind. To bypass difficulties, we propose here an inventory of some tree-like structures for *on-line* problems mainly. We understand by *on-line problems* problems where the structure of data is modified during time. They can involve for example a dictionary, a database where the basic operations (insertion, deletion, search) may arrive at any moment.

We do not intend, in this section, to be interested in the storage of *static data structures* as with the matrix types we have examined previously. However, note that all the structures are balanced: the maximum distance of any leaf from the root of a tree differs by one.

As in algorithmics for disks, the criteria of performance for a data stucture are a combination of the execution time of a search, insert or delete operation, the number of I/O operations necessary for the basic operations and of the storage space of the structure.

We need the following notations: N is the input size of the problem (number of items to deal with), Z is the output size of the problem, M is size of the main memory, B the size of a block, D is the number of independent disks. Furthermore one asks:

$$M < N, \quad 1 \leq D.B \leq M$$

$$n = \frac{N}{B}, \quad z = \frac{Z}{B}, \quad m = \frac{M}{B}$$

The challenges are:

1. to answer requests for a search in $\mathcal{O}(\log_B N + z)$ I/Os;
2. to use a linear space of storage;
3. to do the updates in $\mathcal{O}(\log_B N)$ I/Os.

If one is not careful there, we can easily build data structures that are completely unsuitable for computation with disks. For example, let us suppose that for a problem, its "in-core" solution passes by a binary tree. The CPU time of a search request is in $\mathcal{O}(\log_2 N + Z)$ ... but it requires one I/O operation to cross one node to the other. So, we also have $\Omega(\log_2 N + Z)$ I/Os which is excessive!

The realization of data structures suited for problems requiring disks aims to build locality of data accesses directly in the data structure and manage explicitly the I/O operations instead of letting the operating system do it.

For our previous example of binary tree, one aims to get a relative gain of $(\log N + Z)/(\log_B N + z)$ which is at least $(\log N)/(\log_B N) = \log B$ which is significant in practice.

Basic data structures for disks have been known for 30 years [KNU 98], others were invented recently, for a particular need. We can quote a certain number of structures for the following problems:

1-D Search: B-trees (which are trees with a number of indexes between $B/2$ and $B - 1$);

Batch problems: Buffer trees;

Multi dimensionnel problems: R-trees;

Problems of reconstruction: Weight balanced B-trees.

An important number of on-line problems is well known and well treated for disks, among which (note furthermore that it is about problems in computational geometry [ABE 99]):

1. Proximity queries, nearest neighbor, clustering;
2. Point location;
3. Ray shooting;
4. Calculations of intervals (External inverval trees).

We also encourage the reader to visit http://www.cs.duke.edu/~large for papers from an algorithmic point of view. We now review some data structures.

1.5.1 B-tree and variants

This is a structure known for 30 years [BAY 72]. The properties are following and a B-tree layout is given in Figure 1.7 (in fact it is a picture of a B^+*tree* which we will see later on):

- Every node is arranged in a disk block;
- $\frac{B}{2} \leq$ node degree $\leq B - 1$ (except for the root);
- Internal nodes store keys and pointers to drive the search;
- The operations of insertion, deletion, deletion of the smallest are performed in $\mathcal{O}(\log_B N)$ I/O operations;
- The deletion or the insertion are operations which present certain technical difficulties when one node becomes empty or full because at this moment we have to rebalance the tree to preserve the the tree property and to bound I/O operations. But we know how to treat them as it is illustrated partially in Figure 1.8.

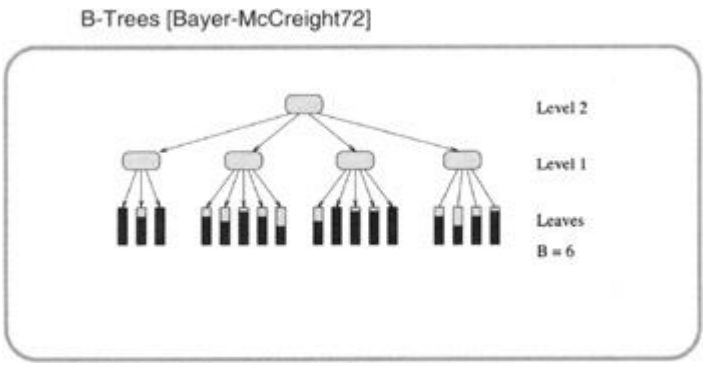


Figure 1.7: B-tree layout

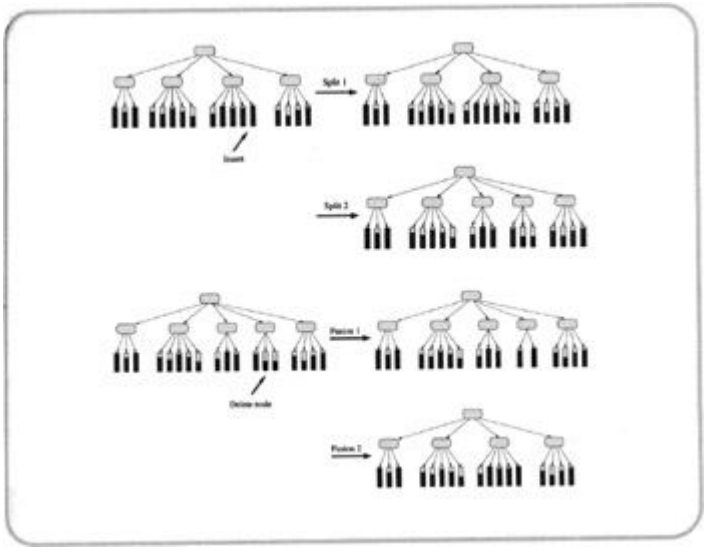


Figure 1.8: Example of re-balancing operations in a B-tree

1.5.1.1 Some data structures close to the B-tree

We shall mention in particular:

B^+-tree: All the items are stored in leaves. Internal nodes store keys and pointers (for a given size one can store "more"). The consequence is that the height of the tree is smaller than for a B-tree and so the search will be faster. Leaves are linked to facilitate "range queries" and/or sequential search;

B*-tree: When one node is full, we have to deal with a question of delaying the rebalancing: we redistribute when two node heights are occupied in 2/3 - for B-tree, one redistributes when only one node becomes full. It is a matter of reducing the frequency of creation of new nodes which is expensive in practice;

Prefix B-tree: Let us consider a B^+-tree with one index of type character strings: one keeps the smallest prefixes as separators and we obtain a greater branching factor;

What is impracticable: Binary B-tree with $B = 2$ and 2-3 trees which are feasible only in main memory.

1.5.2 Buffer-tree and R-tree

These data structures are used in computational geometry and were invented in 1995 by Arge [ARG 95] and in 1984 by Guttman [GUT 84] respectively. An example of R-tree is given in Figure 1.9 and an example of Buffer-tree is presented in Figure 1.10. R-trees are used to store rectangles, lines, points in computational problems. The most important properties of R-trees are the following:

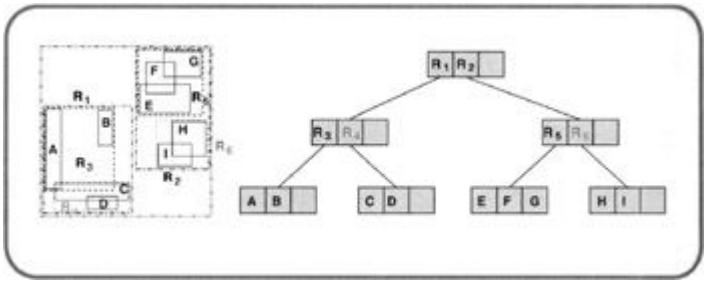


Figure 1.9: Representation of an R-tree

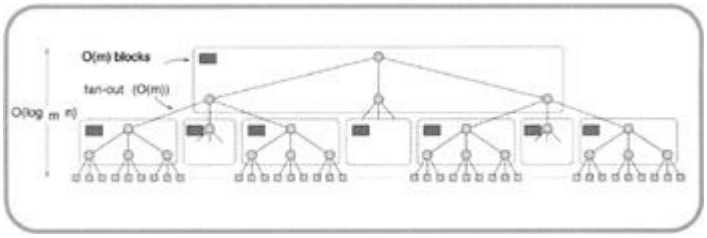


Figure 1.10: Representation of a Buffer-tree

Usability: search in space dimensions superior to 1. They allow one to keep a linear space of storage. The idea is to code the information on *boxes* containing a certain number of rectangles. For example, in Figure 1.9, the root node stores box R_1 defined by points at the "left and right corners" of the box. R_1 contains rectangles A, B, C, D;

Degree of internal nodes: $\Theta(B)$ and leaves store $\Theta(B)$ items. At every node, we associate a box called a *bounding box* with all the elements in sub-trees. So, in Figure 1.9, the R_2 box contains all the points which are in the interior of this box. By considering one node of the tree one is capable of saying if a point is "inside the considered sub-tree" at constant time $\Theta(B)$;

In practice: They have good behavior for small dimensions.

The Buffer-tree structure (see Figure 1.10) was conceived to remedy the following question: What can one do if the requests can arrive in any order? The main idea is the following one: we logically group nodes and we add buffers. So, when a buffer becomes full, items go down from one level. The essential properties of Buffer-tree are the following:

1. It is a well-balanced tree of degree $\Theta(m)$ with a buffer of size M for every node;
2. Items in one node go down from a level when the buffer is full but not when one node becomes full;
3. To empty a buffer, it requires $\mathcal{O}(m)$ I/O's that amortize the distribution cost of M items towards $\Theta(m)$ indexes;
4. For every item, there is an amortized cost of $\mathcal{O}(m/M) = \mathcal{O}(1/B)$ I/Os by level. Search and update operations are realized in $\mathcal{O}((1/B)\log_m n)$ I/Os.

1.5.3 Structures adapted in search of character strings

Character strings are the data structure of preference in bio-informatics. The sequential pattern matching algorithms in main memory [APO 92, BAE 92] and in a lesser measure in parallel was widely studied in the past for numerous problems [CRO 99]; the sequence alignment problem, for example.

Tree-like data structures such as Patricia trees, digital trees (see [pat91] for a synthesis) were proposed 30 years ago. These structures were conceived with the idea to implement them in main memory. In the literature, there are very few data structures, to our knowledge, for the search of strings in external memory. We can quote mainly two studies: Ferragina's String B-Tree [FER 99] and Suffix Trees [CLA 96] of Clark and Munro.

1.5.3.1 The structure of string B-tree

This is a data structure adapted to the storage of arbitrarily long character strings, which, contrary to the known structures (Patricia and Suffixe trees) do not hold in main memory. This data structure should interest the researchers in biology,

crystallography etc.

However biologists, crystallographers store their data in a *flat* way and effective algorithms adapted to solve their problems is not yet met, in our opinion.

To obtain good complexities for the search or the update operations in the worst case, the data structure appears as a combination of a B-tree and a Patricia tree to which one adds pointers. In fact, it is about the first data structure managing arbitrarily long strings which offers the same complexities as B-tree with regards to the elementary operations (search, update...).

1.5.4 Conclusions about tree structures

B-trees and variant are widely used today. However some problems remain open, in particular on the way to optimize cache (buffer) management and data transfers in the case of a multiple disks system. In fact, recent trends in processor design are such that one can build "big caches" in a way that the CPU is not "bounded" by I/O operations but is stalled while servicing CPU data cache-misses.

The paper of Shimin Chen and all. [CHE 02] is of particular interest. The authors propose to employ a special technique named *fractal prefetching* which embed "cache-optimized" trees whithin "disk-optimized" trees. To optimize I/O performance, traditional "disk-optimized" B-trees are composed of nodes of size of a *disk page* i.e. the natural size for reading or writing to disks. "Cache-optimized" B-trees are built to optimize CPU cache performance by minimizing the impact of cache misses.

The technique used in the paper [CHE 02] is based on *prefetching*, which we introduce now as a core technique for I/O performances.

1.6 I/O (automatic) prefetching

We investigate now some challenges of fully automatic techniques to manage the I/O requirements of out-of-core programs. Mainly, in the first subsections the compiler determines information on future access patterns and inserts *prefetch* operations to initiate I/O before the data is needed. No modifications to the original source code and accomplished by the programmer are needed to support the paradigm. All the modifications made by the compiler are hidden to the programmer and to the users. The last subsections are devoted to techniques that do not rely on any compiler but the aim is to optimize buffer management and to propose prefetch schedules.

Prefetching techniques are now implemented in hardware and are well used in many applications to accelerate RAM transfers and/or to diminish cache problems [HEN 02] for in-core applications. For instance, on the Alpha 21164 processor, the *prefetch* instruction exists at the assembly level and any compiler can use it. GCC-3.2 (a x86 free C compiler) is now able to insert prefetch instructions for Athlon/Pentium processors. To be effective, one speculates on what will be the execution by examining the memory access patterns.

After a presentation of alternate techniques (paged virtual memory and explicit I/O), we recall, through an in-core example, how the technique applies and then we can adapt it for I/O requirements.

1.6.1 Paged virtual memory

Another way to bypass code rewriting is to count on the virtual page memory service offered by the operating system. They are three primary reasons why virtual memory systems (if we do not pay attention to them) are unable to provide the best I/O performance.

First, in a paged virtual memory data is brought into memory in response to a *page fault* which occurs when one attempts to read a virtual memory location that is not in physical memory. In the case of a fault, the operating system gives to the CPU another process. The technique improves the throughput of the operating system but does not improve the performance of the faulting process.

Second, page faults involve only a single page-sized read request at a time which is, generally speaking, too small to obtain performance. Third, without knowledge of the application it is difficult to predict which page to evict to make room for the new one.

1.6.2 Explicit I/Os

Rewriting out-of-core applications to use explicit I/O calls can potentially achieve better performance because you may write large disk requests to utilize the available bandwidth, you potentially control the data replacement in main memory and you can allow the overlapping of non-blocking I/O calls with computations.

The main drawback of this approach is that the re-coding-stage is-important and heavy: the programmer should also pay attention to buffer sizes used by I/O buffering and more generally to available resources. Thus, *prefetching*, directed by the compiler, is an intermediate solution to obtain performance and aims to offer better performance than a virtual paged system without the cost of explicitly rewriting a piece of code.

1.6.3 Prefetching in-core programs

Let us consider now the following code (see [VAN 97]) consisting of a prefix-sum computation. It will serve as a toy example along the section:

```
sum = 0;
for (i=0; i<N; i++) sum = a[i]+sum;
```

The problem is to insert *prefetch* instructions to diminish cache problems (misses) [TOR 95]. The assumptions about the cache behavior are the following:

1. a cache block (line) size is 4 machine words; A fetch instruction brings 4 data into the cache;
2. we prefetch one data before its use and we have enough time to accomplish the transfer from memory to the cache;

First of all, elapsed time between the prefetch instruction and the load instruction which follows varies because of the latencies of memories. If the compiler authorizes the fetch too late, the data will not be in the cache; if too early one is going to eliminate a data of cache. The second assumption captures the problem.

Second, and according to our assumptions, our previous code generates a cache miss every four accesses. The improvement follows:

```
sum = 0;
```

```
for (i=0; i<N; i+=4) {
    fetch(&a[i+4]);
    sum = a[i] + sum;
    sum = a[i+1] + sum;
    sum = a[i+2] + sum;
    sum = a[i+3] + sum;
}
```

In order to treat side effects, the code is rewritten as follows:

```
/* First we prepare data: we bring data into cache */
fetch(&sum);
fetch(&a[0]);
/* we iterate and fetch */
for (i=0; i<N-4; i+=4) {
    fetch(&a[i+4]);
    sum = a[i] + sum;
    sum = a[i+1] + sum;
    sum = a[i+2] + sum;
    sum = a[i+3] + sum;}
/ * Epilogue for the last iterations */
for( ; i<N; i++)
    sum = a[i] + sum;
```

Now, what is the problem with the code? If, for our processor the average cache miss latency[5] is about 100 processor cycles while the smallest loop execution time is 45 cycles (due to the computations of the sums), then it is more judicious to reference data $\lceil \frac{100}{45} \rceil = 3$ iterations before their use. In the following code, we prefetch 3 iterations in advance, a fetch instruction is supposed to bring 4 words in memory:

```
/* We prepare cache */
fetch(&sum);
for(i=0;i<12;i +=4).fetch(&a[i]);
for (i=0; i<N-12; i+=4) {

   /* fetch operations are non blocking operations. */
   fetch(&a[i+12]);

   sum = a[i] + sum; sum = a[i+1] + sum;
   sum = a[i+2] + sum; sum = a[i+3] + sum; /* 45 cycles */
}
/ * Epilogue for the last iterations */
for( ; i<N; i++)
   sum = a[i] + sum;
```

So, when a data is accessed we find it necessarily in the cache. We note that we have to focus on the time duration of the loop[6] and to consider the cache miss latency for a particular architecture.

In practice, a fetch operation last about 100 cycles on the Alpha 21164 architecture. We can overlap a computation of 100 cycles with one fetch. Such an overlap represents some fraction of microseconds. Now, if we want to overlap computation and I/O operations, the overlaping should be some milliseconds. To be effective, prefetching I/O operations could be envisaged at a coarse grain task level which means at the procedure level rather than at a fine-grain-level-which means at the instruction level. In fact, I/O prefetching is accomplished at the block level.

1.6.4 Prefetching I/O operations

The goal of prefetching I/O operations is to hide the different I/O latencies:

- *write* latency is easy to hide because they can be buffered or packed to form large requests;
- *read* latency is more difficult to hide because the application stalls waiting for the read to complete. The key to success is to separate the *request* for data and the *use* of that data, while finding enough work to accomplish between the request and its completion.

It is easy to observe that prefetching [MOW 96, DEM 97, KOT 91] does not reduce the number of disk accesses but attemps to anticipate, to re-organize them.

Studies on compiler-based prefetching to hide cache to memory latency [MOW 92] have demonstrated the importance of avoiding the unnecessarily prefetch of data that already resides in cache. An analogous situation exists with I/O prefetching since we do not want to load a page that already resides in main memory.

But it is considerably more difficult to avoid unnecessary prefetches because main memory is much larger than a cache and thus an estimate or a technique used for a better utilization of caches cannot be re-used without profound modifications. In another words, a simple analysis tends to underestimate the ability to retain data in main memory.

The first prefetching algorithm is the well known One-Block-Ahead (OBA) algorithm. It is based on the assumption that frequent I/O operations have sequential locality i.e. after accessing $block_n$, $block_{n+1}$ is requested. The main problem with this algorithm is that in parallel out-of-core algorithms files are not necessary accessed sequentially but they can be accessed using strided or even more complicated patterns [NIE 96, Pur 94].

The OBA algorithm can be used when one expects to see requests, predominantly sequential access patterns. A study on a characterization of the I/O behavior of a commercial distributed-shared-memory machine [BOR 00] concludes that "at present, the I/O subsystems are being designed in isolation, and there is a need for mending the traditional memory-oriented design approach to address this problem". Authors observe that on the commercial clustered DSM machine used, the HP Exemplar, built as a cluster of SMP (Symmetric Multi-Processor) nodes, integrated clustering of computing and I/O resources, both hardware and software, is "not advantageous for two reasons. First, within an SMP node, the I/O bandwidth is often restricted by the performance of the peripheral components and cannot match the memory bandwidth. Second, since the I/O resources are shared as a global resource, the file-access costs become nonuniform and the I/O behavior of the entire system, in terms of both scalability and balance degrades".

1.6.5 Prediction and prefetching

For scientific out-of-core algorithms [NIE 95] alternate prefetching techniques should be devised. In [COR 99], the authors propose a technique to find the access pattern used by the application to predict the block that will be requested in the future. The technique is an adaptation of the Prediction by Partial Match (PPM) technique proposed by Vitter and Krishnan in [VIT 96, CUR 93].

In the algorithm of Cortes and Labata [COR 99] which is devoted to improve file system performance, a sequence of user requests is modelized by a list pairs formed by a *size* and an *offset interval*. The size is the number of blocks in a request. The interval offset is the difference in blocks between the first block of a given request and the first block of the previous one. The algorithm attemps to detect the access pattern of offset intervals and requested sizes. We want to know that after jumping *x* blocks, the following request will jump *y* blocks.

The aim is to predict both the next block to be accessed and the number of blocks to prefetch. The data structure is a graph where offset intervals and sizes are kept in nodes while the links between nodes are labeled with the time they were last used.

We keep the example of [COR 99]. So, we assume that the access pattern is depicted in Figure 1.11. We note that an access of two blocks is always followed by another one of three blocks that is located three blocks ahead. The next block is located five blocks ahead and so on!

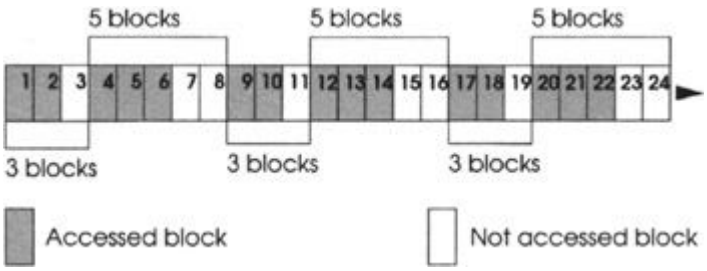


Figure 1.11: Access pattern used in the example

After the first reading, the algorithm cannot do anything because it has not enough information (see Figure 1.11, frame t_1). After the second reading, the data structure is the one-depicted-in-Figure 1.11, frame t_2. The new node contains the information saying that there is a 3-block request after an offset interval of three blocks. The construction after the third reading is given in Figure 1.11, frame t_3. A new node has been created; it specifies a 2-block request after a 5-blocks interval. Furthermore, the first link can also be added. The label (t_3) gives the information that, after the third reading, the algorithm is "in the state" given by the second created node. The construction goes on from frame $t - 4$ until frame t_6.

Once the graph in Figure 1.11, frame t_6 is built, the algorithm uses it to predict the blocks needed by the application. Basically, the prediction corresponds to the reading of *I, S* values of a node from a given node and a time value. The graph can be assimilated to a state automaton and a prediction is made after a draw of a label.

The algorithm of [COR 99] is a little bit more complicated because we also have to deal with the problem of a miss-prediction. Imagine for instance that the seventh reading concerns a 4-blocks reading with a 7-blocks interval. A new node should be added to the graph; the prediction is incorrect; the blocks that have been read are not useful, we must evict them. This is the same problem found in branch predictors that equiped all CPU [HEN 02]. Branch predictors are in charge to fetch in memory the next instruction or data to execute. We do not develop this here and we stay at this level of description.

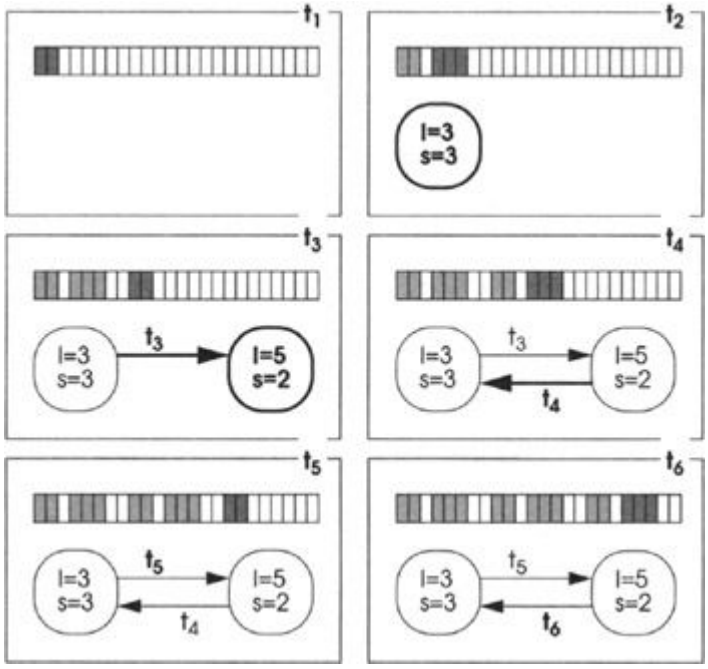


Figure 1.12: Steps of data structure construction for example in Figure 1.11

However we note that other problems occur with the algorithm. First, when there is no node that describes the current situation, there is nothing the system can do to predict the next block. Cortes, in [COR 99] proposes to use the OBA algorithm.

Second, when two different arrows exit from a node, we have a conflict: in which node shall we go? This is a problem of non-determinism. One solution is to follow the most recently updated link. Another heuristic is to predict the same node as previously (the node we have traversed the last time we were in the current node).

A more aggressive prefetching algorithm is described in [COR 99] accompanied by experimental studies. Replacement strategies and buffer management are examined in [VAR 99]. These two last papers constitute a continuation of problems and solutions exposed in this section.

1.6.6 Prefetching and file buffer management

Let us sketch here the work of Hugo Patterson, Garth Gibson, Eka Ginting, Daniel Stodolsky and Jim Zelenka [PAT 95] about prefetching. Since the study is about prefetching, the main idea is to propose "that applications should issue hints which disclose their future I/O accesses".

The authors studied three competing access demands: prefetching hinted blocks, cache hinted blocks for reuse and caching recently used data for unhinted accesses. So, they distinguish and refine the policies of data prefetching based on hints, the management of caches, first when a hint assumes that in a near future we will reuse a block of data and second when the system cannot predict the next access: in this case, it seems reasonable to keep information about recently used blocks and to try to maintain the blocks in the cache because we guess that they will be reused in the future.

Another objective of the work [PAT 95] is to optimize file buffer management. Authors propose an I/O manager that resolves the problems due to the presence of the traditional LRU (Least Recently Used) cache and the cache of hinted blocks. The LRU cache uses the rule that the least recent block is least valuable. The "prefetching cache" of hinted blocks identifies as least valuable the block whose next hinted access is furthest in the future. The main concern is to decide when cache buffers should be allocated for prefetching vs caching for hinted re-use vs. caching into the LRU cache for unhinted accesses.

The role of the *buffer allocator* is, among others functionalities, to allocate cache buffers in order to minimize application execution time. The decisions made by the buffer allocator are made according to a set of cost metrics, for instance the cost of ejecting a hinted block. The different metrics are modelized in the paper [PAT 95]. The technique here uses a system performance model to estimate the benefit of using a buffer for prefetching and the cost of taking a buffer from the LRU cache. The metrics are estimated dynamically and they allow one to reallocate a buffer from the cache for prefetching when the benefit is greater than the cost.

More information about the technique can also be found on Carnegie Melon University web page: http://www.pd1.cmu.edu/TIP/cost_benefit.html.

1.6.7 Prefetching and queued writing

In the article entitled "*Duality between Prefetching and Queued Writing with Parallel Disks*" [HUT 01]. Hutchinson, Sanders and Vitter explored the duality between the problems of writing to parallel disks and the problems of prefetching.

The aim of the work [HUT 01] is to provide simple algorithms for computing optimal prefetch schedules. In fact, the authors studied two problems: the *output scheduling* (or *queued writing*) problem and the *prefetch scheduling* problem.

The *output scheduling* problem takes as input a fixed size pool of m (initialily empty) memory buffers for storing blocks and the sequence $\Sigma = w_0, w_1,\ldots,w_{L-1}$ of block *write requests* as they are issued. So, the goal is to build an output schedule which is specified by a function $\texttt{oStep} : \{w_0,\ldots,w_{L-1}\} \rightarrow \mathbb{N}$ that provides for each block $w_i \in \Sigma$, the time step when it will be output on disk. Each write request is also labeled with the disk it will use. The resulting schedule proposed in [HUT 01] specifies when the blocks are output on disks.

The *prefetch scheduling* problem takes as input a fixed size pool of m (empty) memory buffers for storing blocks and the sequence $\Sigma = r_0, r_1,\ldots,r_{L-1}$ of distinct block *read requests* that will be issued. Each read request is labeled, as previously, with the disk it will use. The *prefetch schedule* proposed in [HUT 01] specifies when the blocks should be fetched so that they can be consumed by the application in the right order.

The duality notion between the two problems could be interpreted in the following manner (quoted from [HUT 01]): "an output schedule corresponds to a prefetch schedule with reversed time axis and vice versa". So, if we derive an (optimal) read-once output scheduling algorithm we apply the *duality principle* to get an (optimal) read-once prefetch scheduling algorithm.

1.6.7.1 Complementary definitions

In order to provide a precise description of the problem, we reuse now the notations and the sequential presentation introduced in [HUT 01].

Let $disk(b_i)$ denote the disk on which block b_i is located. An application has to write blocks according to the order specified by Σ. We assume that the sequence of blocks to write onto disks is given. This may correspond to the situation where disks accesses are totally predictable. We assume also that we have one CPU connected to a pool of (identical) disks. In fact, blocks are written first into buffers attached to disks, then a scheduling algorithm orchestrates the physical output of these blocks to disks. An output schedule is said to be *correct* if the following conditions hold:

i. no disk is referenced more than once in a single time step, i.e. if $i \neq j$ and $\text{disk}(b_i) = \text{disk}(b_j)$ then $\texttt{oStep}(b_i) \neq \texttt{oStep}(b_j)$;

ii. the buffer pool is large enough to hold all the blocks b_j that are written before a block b_i but not output before b_i i.e.

$$\forall i : 0 \leq i \leq L : \text{oBacklog}(b_i) :=\mid \{j < i : \text{oStep}(b_j) \geq \text{oStep}(b_i)\} \mid < m$$

So m represents the maximal number of blocks that fit into the different buffers. Moreover, the number of steps needed by an output schedule is:

$$T = \max_{0 \leq i \leq L} \{\text{oStep}(b_i)\}$$

A schedule is *optimal* if it minimizes T among all correct schedules. Now a prefetch schedule is defined in a similar way and using a function $\texttt{iStep} : \{b_0,\ldots,b_{L-1}\} \rightarrow \mathbb{N}$ that provides for each block $b_i \in \Sigma$, the time step when it will be fetched and stored in the buffers. The limited buffer pool size (m) requires the following correctness condition:

i. all blocks b_j that are fetched no later than a block b_i but consumed after b_i must be buffered i.e.

$$\forall i : 0 \leq i \leq L : \text{iBacklog}(b_i) :=\mid \{j > i : \text{iStep}(b_j) \leq \text{oStep}(b_i)\} \mid < m$$

Then, the main theorem presented in [HUT 01] shows that the reading and writing schedules are equivalent to each other in the following sense:

Theorem: 1 (Duality principle)

Consider any sequence $\Sigma = b_0,\ldots,b_{L-1}$ of distinct write requests. Let oStep denote a correct output schedule for Σ that uses T output steps. Then we get a correct prefetch schedule iStep for $\Sigma^R = b_{L-1}, \ldots, b_0$ that uses T fetch steps by setting $\text{iStep}(b_i) = T - \text{oStep}(b_i) + 1$. *Vice versa, every correct prefetch schedule iStep for Σ^R that uses T fetch steps yields a correct output schedule $oStep(b_i) = T - iStep(b_i) + 1$ for Σ, using T output steps.*

For the proof of theorem 1, please consult [HUT 01]. Note also that we require an input sequence of distinct blocks because in the other case we have the possibility of saving disk accesses by keeping previously accessed block in memory and thus to reduce T. So, the approach with no duplicate facilitates the problem solving.

1.6.7.2 An algorithm for the output schedule

Consider now the following algorithm named *greedy algorithm* in [HUT 01] for writing a sequence $\Sigma = b_0,\ldots,b_{L-1})$ of distinct blocks. Let Q denotes the set of blocks in the buffer pool, so initially $Q =$. Let $Q_d = \{b \quad Q : \text{disk}(b) = d\}$. Write the blocks b_l in sequence as follows:

1. if $| Q |< m$ then simply insert b_l into Q;
2. otherwise, each disk with $Q_d \neq$ outputs the block in Q_d that appears first in Σ. The blocks output are then removed from Q and b_j is inserted into Q;
3. once all blocks are written, the queues are flushed, i.e. additional output steps are performed until Q is empty.

Note that any schedule where blocks are output in arrival order on each disk is called a FIFO schedule. This notion is important in the previous algorithm and proof of the theorem saying that the algorithm provides an optimal output schedule can be devised in [HUT 01].

1.6.7.3 Example

Let us sketch an example to observe how the algorithm yields a correct and optimal output schedule. Let $\Sigma = b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8$. Our disk system is composed of 3 disks d_0, d_1, d_2 and we set $m = 2$. Block distribution should be made according to the following mapping:

$$\begin{array}{ll} \text{disk}(b_0) = 0 & \text{disk}(b_1) = 1 \\ \text{disk}(b_2) = 2 & \text{disk}(b_3) = 1 \\ \text{disk}(b_4) = 2 & \text{disk}(b_5) = 2 \\ \text{disk}(b_6) = 0 & \text{disk}(b_7) = 2 \\ \text{disk}(b_8) = 0 & \end{array}$$

First, we compute the Q sets: $Q_0 = \{b_0, b_6, b_8\}$, $Q_1 = \{b_1, b_3\}$, $Q_2 = \{b_2, b_4, b_5, b_7\}$. Second, in Figure 1.13 we observe that the final output schedule has 10 steps. Note that symbols $\downarrow$ in Figure 1.13 signify that a particular data is flushed to disk at time labeled i).

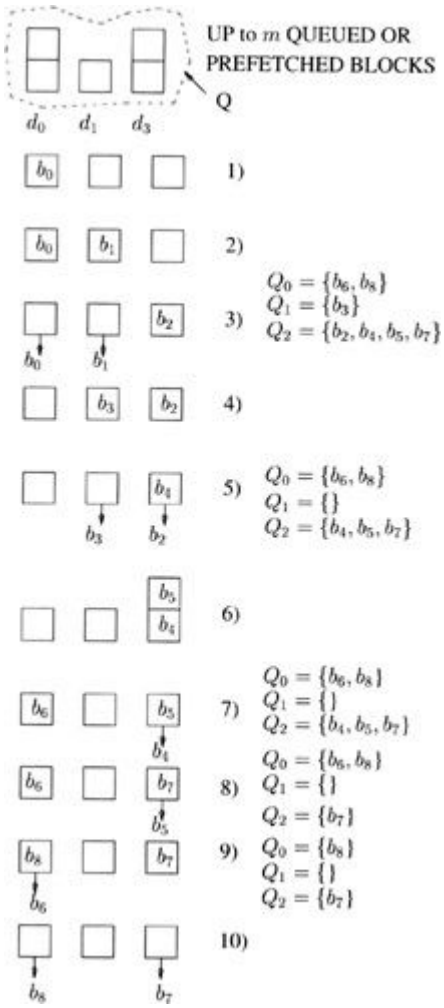


Figure 1.13: Trace execution of suggested example

According to the duality theorem (see Theorem 1) we get immediately an (optimal) prefetch iStep schedule for $\Sigma = b_8, \ldots, b_1, b_0$ which is defined as follows:

$$\begin{aligned}
\text{iStep}(b_0) &= 10-3+1 = 8\\
\text{iStep}(b_1) &= 10-3+1 = 8\\
\text{iStep}(b_2) &= 10-5+1 = 6\\
\text{iStep}(b_3) &= 10-5+1 = 6\\
\text{iStep}(b_4) &= 10-7+1 = 4\\
\text{iStep}(b_5) &= 10-8+1 = 3\\
\text{iStep}(b_6) &= 10-9+1 = 2\\
\text{iStep}(b_7) &= 10-10+1 = 1\\
\text{iStep}(b_8) &= 10-10+1 = 1
\end{aligned}$$

1.6.7.4 Some remarks and further work

In the previous algorithm, when all disk requests concern only one disk, we have no parallelism and the writings are done sequentially! Conversely, when all the writings concern different disks we potentially solve the problem in one step!

So, the question "What is the minimal number of steps that is necessary to process (in parallel and/or in sequential) a given sequence of *n* blocks writings?" is important. Clearly, the minimal number of steps is bounded by 1 and *n*.

The pool can be used to reorder the outputs (when they are known in advance) with respect to the logical writing order given by the alphabet 0,1, ... *L* – 1. In doing this, we certainly do not obtain a *FIFO schedule* since we allow some writings to skip over other writings. For instance, in our previous example (see Figure 1.13), at time 6) we could fill the first disk with b_6 instead of processing b_5. In doing this, we could flush to disk, at step 7) both b_6, b_4 instead of the sole b_4.

In order to formalize the previous idea, we need complementary definitions related to the theory of the *partially commutative monoid* where the objects of the theory are called *traces*. The theory was introduced in 1997 by Mazurkiewicz [MAZ 77] as a model of distributed computation. In fact, D. Foata and P. Cartier [CAR 69] were the first researchers interested by the monoid in order to describe combinatorial properties of words. We do not recall here all the basic definitions of *alphabet*, *word*, *language*... of the formal language theory.

Let Σ be a finite alphabet and $I \subseteq \Sigma \times \Sigma$ be a symmetric and anti-reflexive relation. We say that two letters $a, b \in \Sigma$ commute if $(a, b) \in I$. The two properties express concurrency: the anti-reflexive property says that the letter *a* can not be concurrent with itself and the symmetry property says that the concurrence of actions is mutual. *I* is named the *independence or commutation relation*. It is more convenient to consider the relation of non-commutation $\Sigma \times \Sigma - I$ that we represent by *D*.

For our problem of disk scheduling we interpret *I* as follows: *I* is the set of pairs (b_i, b_j), $i \neq j$ such that b_i, b_j represent blocks to be output to disks and such that $\text{disk}(b_i) \neq \text{disk}(b_j)$.

According to our previous example (see also Figure 1.13) we have here that $D =$
$\{(b_0, b_6), (b_6, b_0), (b_0, b_8), (b_8, b_0), (b_6, b_8), (b_8, b_6), (b_2, b_4), (b_4, b_2),$
$(b_2, b_5), (b_5, b_2), (b_2, b_7), (b_7, b_2), (b_4, b_5), (b_5, b_4), (b_4, b_7), (b_7, b_4),$
$(b_5, b_7), (b_7, b_5), (b_1, b_3), (b_3, b_1), \ldots$ and we omit the pairs $(b_i, b_i)\}$

We note $\equiv_I$ the congruence engendered by the relation $R_I = \{(ab, ba)/(a, b) \in I\}$. Two words *u* and *v* of Σ^* are congruent, $u \equiv_I v$, if there exists a sequence (w_i) of words such that $w_1 = u$, $w_k = v$ and for all $i \in [1..k-1]$, $w_i = w'_i abw''_i$, $w_{i+1} = w'_i baw''_i$ with $(a, b) \in I$.

The *Foata Normal Form* (FNF) is now introduced in order to organize a sequence of words which represents the faster execution of the word in parallel.

Let $\mathcal{C}$ denotes the set of cliques of the graph representing *I*. We associate to each $C \in \mathcal{C}$ the homomorphism *h* defined as follows

$$\begin{aligned}
&h: \mathcal{C}^* \to \Sigma^*\\
&\quad c \mapsto a_1 a_2 \cdots a_n \; si \; C = \{a_1, a_2, \cdots, a_n\} \; with \; a_1 < a_2 < \cdots < a_n
\end{aligned}$$

Definition 1: [Foata Normal Form]

We say that a word $u \in \Sigma^*$ is under Foata Normal Form (FNF) if it can be written under the format $u = \cdot u_2 \ldots u_n$ with:

1. $\forall i \in [1 \ldots n]$, $\exists C \in \mathcal{C}$ such that $u = h(C)$. Every u is called a "block" of the FNF;

2. $_i$ [1…n – 1], a $alph(u_{i+1})$, b $alph(u_i)$ such that (a, b) I;

So, blocks in the FNF consist of cliques of the commutation graph. For a given word v Σ^*, we are looking, in the theory of concurrency, for the "best word": the word with the minimal (parallel) execution time. The word is given by the next Theorem:

Theorem: 2 [CAR 69]

Let v Σ^* a non empty word. There exists exactly only one word under Foata Normal Form noted FNF(v) such that v u

The Foata Normal Form has also one remarkable property: for u Σ^* the number of *elementary steps* (the number of blocks in the sense of the definition of FNF) required to execute u is given by FNF(u).

We restart now our example: we have to schedule nine writing requests according to the dependence relation given above. We also set m = 3. It is not difficult to see that the Foata Normal Form of the input is:

$$FNF(b_0.b_1.b_2.b_3.b_4.b_5.b_6.b_7.b_8.b_9) = (b_0.b_1.b_2)(b_3.b_4.b_6)(b_5.b_8)(b_7)$$

The decomposition means that we can schedule the input sequence in four steps and we cannot do better! Figure 1.14 illustrates the results. So, we obtain an optimal output schedule in the sense that it has a minimal time duration.

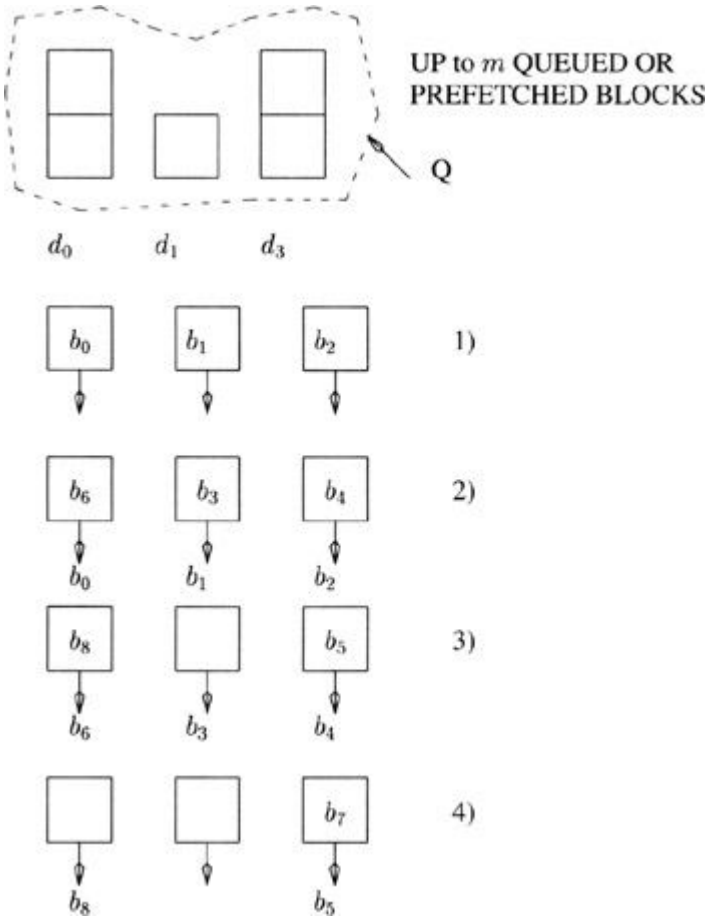


Figure 1.14: Foata Normal Form: schedule of suggested example

1.6.8 Conclusion

We gave a formalization of the scheduling problem when disks are under concern. The first algorithm that we have presented has the property to be a *FIFO schedule*: blocks are output in arrival order. In the second and last part of the section, we have introduced another scheme that allows the reordering of the input. It is the first time, to our knowledge, that the Foata Normal Form (FNF) is used in the context of disk scheduling. It provides also an optimal schedule in the sense that the execution of the input is minimal after the translation into the FNF form.

Many questions are still open when we study scheduling. For instance, can we adapt previous algorithms in the case where the tasks do not last the same amount of time? The general problem is to schedule different tasks (in terms of time duration) on different disks (in terms of disk bandwidth performance or rotational performance).

[5] The time to resolve a miss problem

[6] We have to find a good estimate which is a difficult problem in itself

1.7 Bibliography

[ABE 99] Abello J., Vitter J. S., Eds., *External Memory Algorithms and Visualization*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Providence, RI, 1999.

[APO 92] Apostolico A., Crochemore M., Galil Z., Manber U., Eds., *Combinatorial Pattern Matching*, Berlin, 1992.

[ARG 95] Arge L., *"The Buffer Tree: A New Technique for Optimal I/O-Algorithms (Extended Abstract)"*, Akl S. G., Dehne F. K. H. A., Sack J.-R., Santoro N., Eds., *Algorithms and Data Structures, 4th International Workshop, vol. 955 of Lecture Notes in Computer Science*, Kingston, Ontario, Canada, Springer, p. 334-345, 16-18 August 1995.

[BAE 92] Baeza-Yates R. A., Perleberg C. H., *"Fast and Practical Approximate String Matching"*, Apostolico A., Crochemore M., Galil Z., Manber U., Eds., *Combinatorial Pattern Matching, Third Annual Symposium, vol. 644 of Lecture Notes in Computer Science*, Tucson, Arizona, Springer, p. 185-192, 29 April -1 May 1992.

[BAY 72] Bayer R., Mccreight E. M., *"Organization and Maintenance of Large Ordered Indices"*, *Acta Informatica*, vol. 1, p. 173-189, 1972.

[BOR 00] Bordawekar R., *"Quantitative Characterization and Analysis of the I/O Behavior of a Commercial Distributed-shared-memory Machine"*, *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, num. 5, p. 509-526, May 2000.

[CAR 69] Cartier P., Foata D., *"Problémes combinatoires de commutation et réarrangements"*, *Lecture Notes in Mathematics*, vol. 85, 1969.

[CHE 02] Chen S., Gibbons P. B., Mowry T. C., Valentin G., *"Fractal Prefetching B+trees: Optimizing Both Cache and Disk Performance. To appear in proceedings of SIG-MOD 2002, Madison, Wisc."*, 2002, (See also: http://www.pdl.cmu.edu/Publications/).

[CLA 96] Clark D. R., Munro J. I., *"Efficient Suffix Trees on Secondary Storage (extended abstract)"*, *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, Atlanta, Georgia, p. 383-391, 28-30 January 1996.

[COR 99] Cortes T., Labarta J., *"Linear Aggressive Prefetching: A way to Increase the Performance of Cooperative Caches"*, Soc. I. C., Ed., *Proceedings of International Parallel Processing Symposium (IPPS'99), San Juan, Puerto Rico*, 1999.

[COZ 02] Cozette O., *"Maximisation du débit d'entrées/sorties des grappes de stations par l'utilisation de l'accés distant direct"*, Jemni M., Majoub Z., Trystram D., Eds., *Proceedings of Rencontres francophones du parallélisme (Renpar'14): Hammamet, Tunisia, April 9-13, 2002*, University of Tunis, p. 25-32, 2002.

[CRO 99] Crochemore, HANCART, *"Pattern Matching in Strings"*, *Algorithms and Theory of Computation Handbook*, CRC Press, 1999.

[CUR 93] Curewitz K. M., Krishnan P., Vitter J. S., *"Practical prefetching via data compression"*, p. 257-266, 1993.

[DEM 97] Demke A., *"Automatic I/O Prefetching for Out-of-Core Applications"*, 1997.

[FER 99] Ferragina P., Grossi R., *"The string B-tree: a new data structure for string search in external memory and its applications"*, *Journal of the ACM*, vol. 46, num. 2, p. 236-280, March 1999.

[FOR 94] Forum M. P. I., MPI: A Message Passing Interface Standard, Technical Report num. CS-94-230, University of Tennessee, Knoxville, TX, May 1994.

[GRO 99a] Gropp W., Lusk E., Skjellum A., *Using MPI, Portable Parallel Programming with the Message-Passing Interface* (second edition), MIT Press, Cambridge, MA, USA, 1999.

[GRO 99b] Gropp W., Lusk E., Thakur R., *Using MPI-2: Advanced Features of the Message Passing Interface*, Scientific and Engineering Computation, MIT Press, Cambridge, MA, USA, November 1999.

[GUT 84] Guttman A. *"R-trees: a dynamic index structure for spatial searching"*, *SIGMOD Record (ACM Special Interest*

Group on Management of Data), vol. 14, num. 2, p. 47-57, 1984.

[HEN 02] Hennessy J., Patterson D., *Computer Architecture, A Quantitative Approach (third edition)*, Morgan Kauffmann, 2002.

[HUT 01] Hutchinson D. A., Sanders P., Vitter J. S., *"Duality between Prefetching and Queued Writing with Parallel Disks"*, *Lecture Notes in Computer Science*, vol. 2161, p. 62-72, 2001.

[KNU 98] Knuth D. E., *Sorting and Searching, vol. 3 of The Art of Computer Programming*, Addison-Wesley, Reading, MA, USA, second edition, 1998.

[KOT 91] Kotz D., Ellis C. S., *"Practical Prefetching Techniques for Parallel File Systems"*, *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, IEEE Computer Society Press, p. 182-189, 1991.

[MAZ 77] Mazurkiewicz A., Concurrent Program Schemes and their Interpretations, Report, DAIMI Rep PB 78, Aarhus University, 1977.

[MOW 92] Mowry T., Lam M., Gupta A., *"Design and evaluation of a compiler algorithm for prefetching"*, *ASPLOS-V*, p. 62-73, 1992.

[MOW 96] Mowry T., Demke A., Krieger O., *"Automatic compiler-inserted I/O prefetching for out-of-core applications"*, *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*, USENIX Association, p. 3-17, 1996.

[NIE 95] Nieuwejaar N., Kotz D., Purakayastha A., Ellis C. S., Best M., File-Access Characteristics of Parallel Scientific Workloads, Report num. TR 95-263, Dept. of Computer Science, Dartmouth College, August 1995.

[NIE 96] Nieuwejaar N., Kotz D., Purakayastha A., Ellis C. S., Best M., *"File-Access Characteristics of Parallel Scientific Workloads"*, *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, num. 10, p. 1075-1089, October 1996.

[PAT 95] Patterson R. H., Gibson G. A., Ginting E., Stodolsky D., Zelenka J., *"Informed Prefetching and Caching"*, *15th Symposium on Operating Systems Principles*, ACM, p. 79-95, 1995, (See also: http://www-2.cs.cmu.edu/~garth/).

[pat91] *Computer Algorithms: Key search Strategies*, IEEE Computer Society, technology series, 1991.

[Pur 94] Purakayastha A., Ellis C. S., Kotz D., Nieuwejaar N., Best. M., Characterizing Parallel File-Access Patterns on a Large-Scale Multiprocessor, Report num. Technical report DUKE-TR-1994-33, Department of Computer Science, Duke University, 1994, Printed copies available from T.R. Librarian, Dept. of Computer Science, Duke University, Box 90129, Durham, NC 27708-0129.

[STR 69] Strassen V., *"Gaussian Elimination is Not Optimal"*, *Numerische Mathematik*, vol. 13, p. 354-356, 1969.

[TOR 95] Torng H., Vassiliadis S., *Instruction-level Parallel Processors*, IEEE Computer Society Press, 1995.

[VAN 97] Vanderwiel S., Lilja D., *"When Caches aren't Enough: Data Prefetching Techniques"*, *IEEE Computer*, vol. july, p. 23-30, 1997.

[VAR 99] Varman P. J., Verma R. M., *"Tight Bounds for Prefetching and Buffer Management Algorithms for Parallel I/O Systems"*, *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, num. 12, p. 1262-1275, 1999.

[VIT 96] Vitter J. S., Krishnan P., *"Optimal prefetching via data compression"*, *Journal of the ACM*, vol. 43, num. 5, p. 771-793, 1996.

Chapter 2: Parallel Sorting on Heterogeneous Clusters

Christophe Cérin, Université de Picardie Jules Verne, LaRIA, 5 rue du moulin neuf, 80000 Amiens, France
Hai Jin, Huazhong University of Science and Technology, Wuhan, 430074, China

2.1 Introduction

The general objectives pursued in sequence as in parallel are either the minimization of the number of movement of heads or the number of times we access the disks. Indeed, the costs of access to disks (in particular for architecture of cluster type) is important with the partially mechanical current technologies. For example, the average rate of transfer from the disk of a PC is of the order of 15M bytes per second with an Ultra ATA/66 interface while it is possible to exchange with a Myrinet network[1] at rates of 120M bytes per second!

In order to capture the influencing factors that drive disk performance, we can use a formal disk model or we can build a simulator. From a practical point of view, the following WEB sites point to simulation tools of disk drives. On the sites, we will find there code sources allowing the experiment:

http://www.cs.dartmouth.edu/~dfk/diskmodel.html A Detailed Simulation Model of the HP 97560 Disk Drive;

http://www.ece.cmu.edu/~ganger/disksim/ The DiskSim Simulation Environment.

From a disk model point of view, Vitter's papers [VIT 94a, VIT 94b, AGG 88] make reference as well as the synthesis [ABE 99]. In these articles, one decides the properties of the disk systems in a model which one calls PDM for *Parallel Disk Model*. The disk model is made according to the following parameters:

N = problem size;

M = size of the internal (RAM) memory;

B = size of a transferred block;

D = number of independant disks;

P = number of processing elements.

with $M < N$, and $1 \leq D\ B \leq M/2$ for practical reasons to correspond best in "real world systems". Figure 2.1 describes the PDM model. On the left, a single processor reaches the shared memory and three disks can work in parallel ("à la RAID") but the aforesaid papers of Vitter do not clarify it. In Figure 2.1, right-hand side, processors reach their local disks and can exchange their data contained on the disks only via the interconnection network.

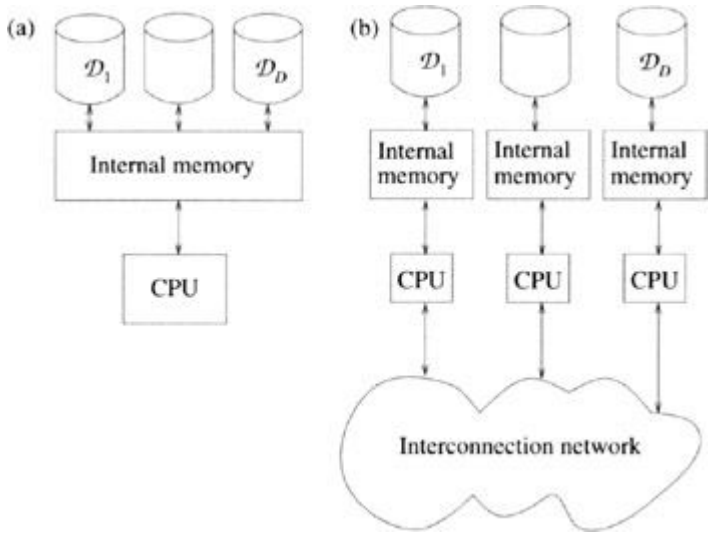


Figure 2.1: See also [VIT 94a]. Models of disks: (a) P = 1. for which D disks are connected to a common processor (b) $P = D$, for which D disks are connected to a different processor. This organization corresponds to the cluster organization

This part of the figure describes the architecture of cluster type. The metrics of performance are defined by considering the number of input communications [MAY 00] between the internal memory and the external disks. One can also be interested in the disk space used by the algorithm as well as in CPU time.

During I/O operation, each of the D disks can simultaneously transfer a block of B consecutive items on the disk. It is useful to consider the following shortcuts:

$$n = \frac{N}{B}, \quad m = \frac{M}{B}$$

The value n represents the number of necessary inputs / outputs to read the input of the problem while m denotes the number of data which holds in main memory. The hypothesis that the given N data $x_1, \cdots, x_N$ are initially distributed (one says "striped data") on D disks is shown in Table 2.1 where we have $D = 4$, $B = 2$.

Table 2.1: Initial data layout with $N = 32$, $D = 4$, $B = 2$

	D_0	D_1	D_2	D_3
strip 0	x_1,x_2	x_3,x_4	x_5,x_6	x_7,x_8
strip 1	x_9,x_{10}	x_{11},x_{12}	x_{13},x_{14}	x_{15},x_{16}
strip 2	x_{17},x_{18}	x_{19},x_{20}	x_{21},x_{22}	x_{23},x_{24}
strip 3	x_{25},x_{26}	x_{27},x_{28}	x_{29},x_{30}	x_{31},x_{32}

Other distributions are possible but the important point is that a distributed file of N data can be read or written in $\mathcal{O}(N/DB) = \mathcal{O}(N/D)$ I/O's operations which is optimal.

The technique known under the name of *disk striping* [SAL 86, KIM 86] authorizes I/O's operations only on whole strips, a strip at a time. The idea is to distribute the first block of a file on the first disk, the second block on the second disk etc. For example, in Table 2.1, the data x_{18}, x_{22} can be reached with a single I/O operation because they are situated on the same strip. So, a striped model with D disks can be considered as only one logical disk with one size of blocks of DB.

Furthermore and in an ideal way, the algorithms using disks should use a linear space of storage i.e. $\mathcal{O}(N/B) = \mathcal{O}(n)$ blocks of disks. It is possible to show that the bound on I/O's operations to sort $N = n$ data with $D \geq 1$ disks is:

$$Sort(N) = \Theta\left(\frac{N}{DB}\log_{M/B}\frac{N}{B}\right) = \Theta\left(\frac{n}{D}\log_m n\right)$$

Note that in practice the term $\log_m N$ is a small constant. The main theorem concerning sorting in the PDM model is then the following one:

Theorem: 3 ([AGG 88], [NOD 95])

The average and worst-case number of I/Os required for sorting $N = nB$ data items using D disks is:

$$Sort(N) = \Theta\left(\frac{n}{D}\log_m n\right) \qquad [2.1]$$

Let us note that in equation 2.1, it is the term $\frac{n}{D} = \frac{N}{B.D}$ that is of importance in practice. So, in order to decrease the number of I/O operations, we need to set B with the highest possible value.

Among the known and employed techniques to accelerate the performances of disk accesses, the technique of *disk striping* can be used to convert in a automatic way an algorithm conceived to use a single disk of block size DB in an algorithm which uses D disks each of size B. It is known that the technique of *disk striping* is not relevant in the case of external sorting.

To observe why it is not relevant, as it is done in [ABE 99], it is necessary to consider the optimal number of I/O operations when one uses a single disk with block sizes equal to B, which is:

$$\Theta(n\log_m n) = \Theta\left(n\frac{\log n}{\log m}\right) = \Theta\left(\frac{N}{B}\frac{\log(n/D)}{\log(M/B)}\right) \qquad [2.2]$$

With the technique of disk striping, the number of I/O operations is the same as if we use blocks of size $D.B$ in the previous optimal algorithm. By replacing B by D B in the equation 2.2 one obtains the bound:

$$\Theta\left(\frac{N}{DB}\frac{\log(n/DB)}{\log(M/DB)}\right) = \Theta\left(\frac{n}{D}\frac{\log(n/D)}{\log(M/D)}\right) \qquad [2.3]$$

Moreover, the bound of external sorting with *D* disks is:

$$\Theta(\frac{n}{D}\log_m n) = \Theta\left(\frac{n}{D}\frac{\log n}{\log m}\right) \qquad [2.4]$$

We notice that the bound given by equation 2.3 is bigger than the bound given by 2.4 by a factor of:

$$\frac{\log(n/D)}{\log n}\frac{\log m}{\log(m/D)} \sim \frac{\log m}{\log m/D}$$

When *D* is of the order of m the term log(*m*/*D*) of the denominator is small and the overhead is of the order of log *m* what can be significant in practice. So, we need special techniques other than the technique of disk striping for external sorting.

[1] http://www.myri.com

2.2 What is an heterogeneous cluster?

The previous discussion was conducted in the context either of sequential sorting or sorting on homogeneous clusters, meaning clusters based on identical mother boards, identical disks, identical CPUs.

We are now interested in a particular case of *non homogeneous clusters* that we define here as usual clusters (network based on a unique communication layer, same installed operating system, same CPU, same memory size and disks) but microprocessors can have *different speeds*. It is a first level of heterogeneity and it introduces new challenges.

In a general way, in a parallel system the notion of heterogeneity touches potentially all the constituents of the system, concerning the following factors:

- The network: the notion of heterogeneity concerns the possibilities of using different media of communication, various methods of signaling, various interfaces and protocols;
- The memory: the notion of heterogeneity concerns the possibilities of using the local memory as well as the distant memory, of managing various levels of hierarchy with various policies of management;
- The processors: the notion of heterogeneity concerns the possibilities of using processors of various manufacturers, processors with various speeds, processors of different internal architectures (RISC (reduced instruction set computer), VLIW (Very Long Instruction Width architecture), multithreadead architecture);
- The software running on the systems: the notion of heterogeneity concerns the possibilities of using several operating systems or different binary codes of the same program;
- The disks: the notion of heterogeneity concerns the possibilities of using several file systems, different media of storage (hard disks, floppy, cartridge), various protocols (IDE ATA, SCSI, PCMCIA).

The parallel problems in an heterogeneous environment, as we have just defined it, establish interesting and particularly useful problems if the user of a cluster can not change all the processors of the cluster but should compose with several versions of one processor, with various speeds. It also seems to us that there is not a lot of literature (in our knowledge) for this class of architecture, many articles (on sorting) on clusters treat homogeneous cases only.

As for the problem to measure effectively the relative speed of processors, we assume the existence of precise techniques to realize it.

2.3 Related works about sorting on clusters

To sort, in parallel, records from which the keys come from an "ordered set" is a problem studied for many years [AKL 85]. It is well known [AKL 85] that from 25 to 50% of all the work carried out by computers consists of sorting. One of the reasons is that it is often easier to manage sorted data than unsorted. For example, the search problem is dealt more quickly if the starting data is sorted.

The studies of parallel sorting algorithms are also guided by necessary properties of the input and the output of the algorithm in order to collect the new architectural paradigms well under a broad pallet of situations. For example, one wants to be interested in the case of *stable sorting* (the order of the equal keys is preserved in the output vector) or with the case where the input is under a particular distribution, for example a cyclic distribution.

Moreover, sorting is particularly interesting (and difficult) because of the irregular access to the memory and bacause of irregular communication patterns. For this reason we think that parallel sorting is a good problem to evaluate parallel machines at the user level.

When researchers study performance, the parallel execution time is not the only center of interest. Blelloch in [BLE 91] defines the concept *sublist expansion metric* as being the ratio between the size of the largest list treated by a processor at one moment of the algorithm to the average expected size. In other words, this metric accounts for load balancing: ideally, a value 1 is required. In this case load balancing is optimal.

Sorting is also present in several test sets which one can find on the site of NASA[2]. The NOW[3] in Berkeley is without any doubt the first project (1997) concerning sorting on clusters. It is interested in a homogeneous cluster of SPARC processors and in the performances in time only. In this work, one is not interested in the problems of load balancing. The project NowSort currently revolves around the project Millenium[4] which proposes to develop a cluster of clusters at the campus of Berkeley in order to support applications of scientific computation, simulation and modeling.

We also find people from industry that are specialists in sorting. For example, the `Ordinal` company[5] distributes Nsort for various platforms, generally multiprocessors machines. For example, in 1997, an Origin2000 with 14 R10000 processors at 195Mhz, 7GB of RAM and 49 disks was able to sort 5.3GB of data in 58 seconds.

The goal is to sort as much data as possible in one minute (minute sort). However, the sorting algorithm has little interest from a scientific point of view: it is a question of reading the disks, of sorting in memory and writing the results on disks. It is brutal, but effective! It will be noted that the total size of the main memory is always greater than the size of the problem!

In 2000 the Nsort algorithm of the Ordinal company sorted 12GB in 1 minute on a SGI ORIGIN IRIX equiped with 32 processors[6]. We also found that sorting a terabyte of 100 bytes records requires 17 minutes and 37 seconds[7] by using the following material solution: "SPsort was run one year RS/6000 SP with 488 nodes. Each node contains 4 332MHz 604 processors, 1.5GB of RAM, and has 9GB SCSI disk. The nodes communicate with one another through the high-speed SP switch which has bi directional link bandwidth to each node at 150 megabytes/sec. Total storage of 6 TB of disk storage in the form of 336 RAID arrays is attached to 56 of the nodes. Besides the 56 disk servers, 400 of the SP nodes actually ran the leaves program".

One is far from the PC at 1000$USD! The record for year 2001 is the following for the Minute test (which consists of sorting as much as possible in a minute): 12GB in 60s, the method used being Ordinal Nsort, on a SGI 32 CPU, Origin IRIX. For the Datamation test (introduced in 1985), to sort a million records of 100 bytes, the record is 0.48s on 32 * (2 * 550*Mhz*) Pentium P3 equipped with 896MB of RAM, 5 * 9*GB* of IBM SCSI disks and a Gigabit Ethernet network.

In 1986 one needed approximately 26s on a CRAY. The complete results are available on the page:

http://research.microsoft.com/barc/SortBenchmark/

[2]See: http://www.nas.nasa.gov/

[3]See the project on: http://now.cs.berkeley.edu/NowSort/index.html/

[4]See: http://www.millennium.berkeley.edu/

[5]See: http://www.ordinal.com/

[6]See the advertisement on http://research.microsoft.com/barc/SortBenchmark/

[7]See: http://www.almaden.ibm.com/cs/gpfs-spsort.html

2.4 Sampling strategies for parallel sorting

The strategy of sampling (pivots) is studied in this section mainly for the case of homogeneous clusters. It is a starting point before explaining our algorithm for external sorting on heterogeneous clusters.

Intuitively it is a question of insulating in the input vector the pivots which would partition it in segments of equal sizes. This phase selects pivots so that between two consecutive pivots, there is the same number of objects. After a phase of redistribution of the data according to values of the pivots, it no longer remains to sort the values locally.

We now explain how this strategy is interesting a priori for architectures of clusters. We explain then why the PSRS (Parallel Sorting by Regular Sampling) technique [REI 87, REI 83, SHI 92, LI 93, HEL 96] is relevant to sorting on homogeneous clusters.

2.5 A short classification of parallel in-core sorting algorithms

It is recognized now that there are two main generic approaches for sorting in parallel. These two approaches appeared relevant in practice: the algorithms implemented according to these techniques are effective on a broad range of architectures. These two approaches are described in the following way:

Merge-based: the algorithms in this family follow the following scheme: (1) each processor initially contains a portion of the list to sort (2) one sorts the portions and one of them exchanges between all the processors (3) one merges the portions in one or more stages;

Quicksort-based: the algorithms in this family follow the following stages: (1) the unsorted lists are partitioning gradually in smaller one, under lists defined by a choice of pivots (2) one sorts lists for which the processors are responsible.

We were interested only in merge-based approach and for this approach only with the algorithms with only one stage of merging. That is thus justified mainly because only one stage of fusion generates very few communications and few overheads due for example to latencies of communications on local area networks of the Ethernet type.

Such a technique is in harmony with what is currently required in programming by message passing: a limited number of long messages. Moreover, when the elements are moved from a processor to another one, they necessarily go to their final destination. References [QUI 89, LI 93, SHI 92, LI 94] belong to the category of *parallel algorithms with only one phase of fusion*; references [BAT 68, BLA 93, BOR 82, COL 88, LEI 84, NAS 82, PLA 89, TRI 93] with those of the *multi steps merge-based algorithms*.

Among all the techniques, the most interesting one for our purpose is, in our opinion, regular sampling such as we will now see. It exceeds on many points all the algorithms previously quoted.

2.5.1 The regular sampling technique

Let us consider that n denotes the size of the problem and p the number of processors. The four canonical stages of the algorithm of sorting by sampling are the following. It is the algorithm of Shi [SHI 92] called PSRS which was conceived for the homogeneous case:

Step 1: one starts by sorting the n/p data locally, then each processor selects p pivots which are gathered on processor 0 (one passes the details concerning the choice of pivots itself).

Step 2: sort on processor 0 of p^2 pivots; we keep $p - 1$ pivots (they are selected at $ip + p/2$, $(1 \leq i \leq (p - 1)$ intervals). All the pivots are broadcast to the other processors.

Step 3: each processor produces p sorted partitions according to the $p - 1$ pivots and sends the partition i (marked by the pivots k_i and k_{i+1}) to the processor i.

Step 4: the processors received sorted partitions; they merge them.

Figure 2.2 page 72 gives an example of unfolding such an algorithm. One notices the initial local sorting on each processor, the choice of the pivots is done at regular intervals, the centralization of the pivots and their sorting and finally the redistribution of the data according to values of the pivots.

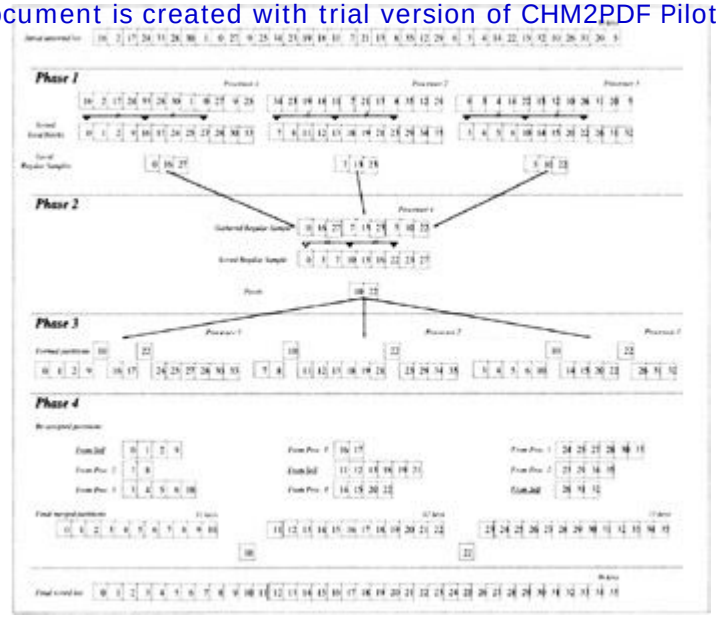


Figure 2.2: Example of PSRS execution [SHI 92]

What makes the strength of this algorithm it is that by sampling on all the processors, we can consider that information on the order of the data is captured. It was shown by Shi [SHI 92] that the computation cost of PSRS matches the optimal bound of $\mathcal{O}(n/p \log n)$. To obtain this result, it is necessary to be ensured of the uniqueness of the data. In this case, one can show that, under the assumption that $n > p^3$, PSRS guarantees a load balancing which differs from optimal by a constant of 2, which is very good! Indeed, that means that at stage 4 of the algorithm presented above, none the processors sorts more twice than it had at the beginning. In practice, the constant is rather close to the optimal one.

2.5.2 A general framework for sorting on heterogeneous clusters

We summarize here the framework of [CÉR 02] for (in-core or out-of-core indistinctly) sorting when the processors do not go at the same speed. The problem is presented as follows: n data (without duplicates) are physically distributed on p processors. Here the processors are characterized by their speeds s_i, ($1 \leq i \leq p$). The rates of transfers with the disks as well as the bandwidth of the network are not captured in the model.

Moreover we are interested in the "*perfect case*" i.e. for the case where the size of the problem can be expressed as p sums. The concept of *lowest common multiple* is useful in order to specify the things here in a mathematical way. In other words, we ask so that the problem size n be expressed in the form:

$$n = k * perf[0] * \mathrm{lcm}(perf, p) + \ldots + k * perf[p-1] * \mathrm{lcm}(perf, p) \quad [2.5]$$

where k is a constant in $\mathbb{N}$, *per f* is a vector of size p which contains the relative performances of the p the processors of the cluster, lcm(*per f*,p) is the smallest common multiple of the p values stored in the *per f* vector.

For example, with k = 1, *per f* = {8, 5, 3, 1}, we describe a processor which runs 8 times more quickly than the slowest, the second processor is running 5 times more quickly than the slowest processor, the third processor is running 3 times more quickly than the slowest and we obtain 1cm({8, 5, 3, 1}, 4) = 120 and thus n = 120 + 3 * 120 + 5 * 120 + 8 * 120 = 2040 is acceptable.

Thus, with problem sizes in the form of equation 2.5, it is very easy for us to assign with each processor a quantity of data *inversely proportional at its speed*. It is the intuitive initial idea and it characterizes the precondition of the problem. The property can be also expressed by the fact that one asks that the size of the problem be divisible by the sum of the values of the `perf` vector. If n cannot be expressed itself as with equation 2.5 , different techniques as those presented in [A.S 95] can be used in order to ensure balancing.

2.5.3 The spring of partitioning

The key point to obtain good performances for load balancing is again the choice of the pivots. Indeed it is these which allow the partitioning of the input in portions of size roughly identical in the homogeneous case. Here, for the heterogeneous case it is necessary that the pivots be arranged so that they constitute portions of sizes proportional to the speed of each processor.

Let us look to Figure 2.3 to explain, without considering too complex technical details, what are the deep springs of partitioning for the heterogeneous case. In Figure 2.3 we have three processors which have three different speeds represented by three different rectangles of different dimensions. The pivots are taken with regular intervals here within the meaning of PSRS and which are represented by the $\leftrightarrow$ symbol.

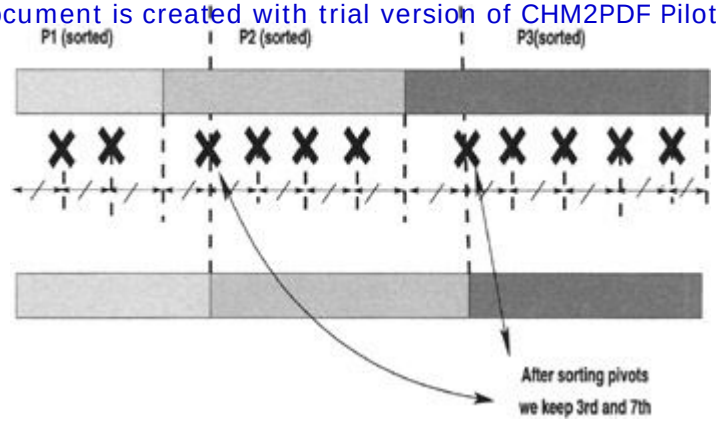


Figure 2.3: Springs of partitioning on heterogeneous clusters

The pivots in Figure 2.3 come from the three processors and they are supposed to be sorted. We refrain here, in particular the third pivot (on the basis of the left part of the figure). The intuitive justification is as follows: in operating that way, we notice that the number of data in the left of this first pivot is necessarily smaller than twice the size of the segment of data contained initially on the left processor (the slower one). While maintaining this invariant for the choices of the other pivots, one arrives at a result of the PSRS type: none of the processors treats more than twice what it had at the beginning.

In fact, we adapt the proof of the PSRS algorithm which is based on the same observations. We obtain a general framework, based on sampling and which offers guarantees in term of load balancing. Consequently, a broader range of architecture of cluster can be used for sorting efficiently on heterogeneous clusters.

2.5.4 The adaptation of the framework for external sorting

Among the most recent articles on parallel external sorting which comprise experimental studies we can quote the references [RAJ 98], [COR 97], [PEA 99]. We will concentrate on some algorithms which are based on sampling techniques. To obtain the bound in equation 2.1 the techniques known under the name of *distribution* or *by fusion* can be examined there too. With these techniques, the *D* disks run in an independant way during parallel reading operations, but in a "striped" way during writing phases.

Let us examine for example the case of the *distribution sort* of Knuth [KNU 98] which is very close, in spirit, to sampling algorithms that we examined in the previous paragraphs. The sorting by distribution scheme (within the meaning of Knuth) is a recursive algorithm for which the entries are successively partitioned by $S - 1$ pivots to form S buckets. The buckets thus formed are in their turn sorted in a recursive way. There is $\log_S(n)$ levels of recursion in the algorithm and the sizes of the pieces decreases by a factor $\theta(S)$ of a level of recursion to another.

If each level uses $\Theta\left(\frac{N}{DB}\right) = \Theta\left(\frac{n}{D}\right)$ operations of I/Os, sorting by distribution has a complexity in a number of I/Os of $\mathcal{O}\left(\frac{n}{D}\log_m n\right)$ which is optimal.

Obviously the key to success depends on the pivots which must divide the pieces in about equal buckets. As that is noted once again in [VIT 94a], "It seems difficult to find $S = \Theta(m)$ splitters using $\Theta(n/D)$ I/Os (the formation of the buckets must be done within this bound to guarantee an optimal algrithm) and guarantee that the bucket sizes are within a constant factor of one another".

The parallel algorithm which in spirit is closest to the algorithms by sampling such as PSRS is for the model with *D* disks, the paper of DeWitt and Al [DEW 91] which looks like a random algorithm with two phases. The algorithm builds small sorted sequences on disk which it is a question of amalgamation in the second phase.

2.5.5 The main algorithm for sorting on heterogeneous clusters

We can also follow the four phases of algorithm PSRS in the following way within the framework of external sorting. Since the first stage of PSRS is a sequential sorting stage we implement it here by a "Polyphase Merge Sort" [KNU 98]. This type of sort is known to use $2m$ intermediate files in order to obtain a "$2m - 1$ way merge" without needing a separate redistribution of data after each pass. In fact it has the advantages of a not balanced sorting without the disadvantage of the redistribution. The algorithm is as follows:

Algorithm 1 (A PSRS scheme for external sorting on heterogeneous clusters)

Preconditions and initial configuration: *we deal with a cluster of* $p = 2^q$ *heterogeneous nodes. The heterogeneous notion is coded in the array 'perf' of size p of integers that denote the relative performances of the p nodes in the machine. The input size is n and n verifies equation 2.5. Initially, disk (or processor) i has* l_i*, a portion of size* $(n/(\sum_{i=1}^{p}$ *per f[i])) * per f[i] of the unsorted list l.*

Step 1: (sequential sorting) *in using polyphase merge sort we get* 2.*l* (1 + log *l*) *10 operations on each processor*

(since we have one disk attached per processor).

Step 2: (selecting pivots) *a sample of candidates is randomly picked from the list. Each processor i picks L = (p – 1) * per f[i] candidates and passes them to a designated processor. This step requires L IO operations that is very inferior, in practice, to the IO operations of step 1. These candidates are sorted and then p – 1 pivots are selected by taking (in a regular way) s^{th}, $2.s^{th}$, …, $(p-1)^{th}$ candidates from the sample. Equipped with equation 2.5, it can be checked that* $s = k * \left\lfloor \frac{n * perf[i] * lcm(perf, p)}{p} \right\rfloor$. *The selected pivots d_1, d_2, …, d_{p-1} are made available to all the processors. All these computations can be carried out in-core since the number of pivots is very small in practice (it is not an order of the internal memory size).*

Step 3: (partitioning) *since the pivots have been sorted (and fit in the main memory), each processor performs binary partitioning on its local portion. Processor i decomposes li according to the pivots. It produces p sublists (files) per processor. Since there are Q = $(n/\sum_{i=1}^{p}$ per f[i]$)$*per f[i] data on processor i, there is no more than 2 * Q/B IOs per processor to accomplish the work in this step (we count read and write of data).*

Step 4: (redistribution of the sublists): *we form messages in order that the message sizes can fit in the local and distant memory. The size is also a multiple of the block size B. If we have an hardware which is able to transfer data from disk to disk, it will be more efficient. The number of IOs is no more than 2 * l_i/B (we also count read (sender side) and write (receiver side) of data).*

Step 5: (final merge) *each processor has in its local disk the p final portions (files) that can be merged with an external merged algorithm for mono-processor system. We re-use the* `mergeSort()` *procedure of the polymerge sort algorithm used in step 1. The number of IO operations is less than* $2.l_i(1 + \log_m l_i)$. *Note that the constant 2 here stands for the bound of data given by the 'PSRS Theorem' that guarantee that in the last step of the algorithm, no processor has to deal with more than twice the initial amount of data. This theorem is still true for the heterogeneous case (see [CÉR 00]) and we apply it.*

Concerning the processor *i* which at the beginning has l_i data, the optimal bound $\Theta\left(\frac{l_i}{B}\log_m \frac{l_i}{B}\right)$ parallel operations of I/Os is reached with this algorithm.

Now let us give others indications about the proof of the result and the way we have implemented it. Our strategy to obtain the same bound as that of PSRS is to place ourself under the conditions of PSRS amgorithm. The technical point which is the most delicate relates to the number and the way of choosing the pivots after the initial sorting.

Since the inputs verify equation 2.5, we can take the pivots in a regular way (within the meaning of PSRS) and in a way proportional to the lowest common multiple of the performances.

From a programming point of view, the code which each processor carries out to select the pivots is this one (the variable `blocksize` is the number of inputs locally to a processor - note also that the value of i is the same on all the processors; this is to conform with equation 2.5):

```
i = (int) (blocksize / (performance[mypid] * nprocs)) - 1;
off = i + 1; k = 0;
while (i <= (blocksize - off - 1)) {
    fseek(MYfpIN, (long) (i * sizeof (MPI_INT)), SEEK_SET);
    fread (&pivot[k], sizeof(MPI_INT), 1, MYfpIN);
    k++;
    i += off;
}
```

The consequence is that one takes a number of pivots proportional to the values of the vector of performance but one makes sure that between two consecutive chosen pivots there is the same number of elements (sorted). This is the essential property of PSRS. Our approach is thus a generalization of PSRS.

2.5.6 Experimentation

The results of experimentation presented in this paragraph were obtained starting from our implementation which can be obtained at URL:

http://www.laria.u-picardie.fr/~cerin/=paladin/

Concerning the thorny question of knowing how one chooses to fill the `perf` vector which gives the relative performances of each node of the cluster, we choose to proceed like this:

- for a problem size *N* and on *p* processors, we carry out tests of the execution time by means of the implementation of

the sequential external sorting (on N/p data) which is used above in the parallel external version of the sort. The ratios at the smallest average execution time obtained give the values of the 'perf' vector.

Thus we collect at the same time the computing power of each CPU but also the performances of the I/O operations. We do not claim that the measured values account for the performances of the nodes in an absolute way but only for the case of sorting.

However, one can say that the main work carried out by our algorithms consists in unrolling sequential versions of sorting, put aside the phases of selection of pivots, pivots sorting in memory and exchange of the partitions.

In Table 2.2 we have a synthetic sight of the architecture of the small cluster with which we experiment with the external sorting algorithm. The `work` partition is useful for all the files created and it corresponds to a SCSI disk.

Table 2.2: Configuration: 4 Alpha 21164 EV 56, 533Mhz - Fast Ethernet

Node	Cache L3/L2/L1	Disk	Linux Kernel	Size of `/work`
helmvige	4MB/96KB/8KB	4GB SCSI	2.2.13 - 0.9	1GB
grimgerde	4MB/96KB/8KB	4GB SCSI	2.2.13 - 0.9	4GB
siegrune	4MB/96KB/8KB	4GB SCSI	2.2.5 - 16	4GB
rossweisse	2MB/94KB/8KB	8GB SCSI	2.2.5 - 16	4GB

In Table 2.3 we have the results of experimentation with the external sequential sorting which uses the algorithm *polyphase merge sort* which is also the sequential brick used in the parallel algorithm. Let us notice that a size of problem of 33554432 integers corresponds to 33554432 * 4 = 134217728 bytes, that is to say 134 *M B*.

Table 2.3: External sequential sorting on our cluster (cf. Table 2.2)

Size	Mean Exe. Time (s)	Stand. deviation
Helmvige		
2097152	22.92146	0.45283
4194304	51.17832	1.99283
8388608	111.40898	1.48268
16777216	235.74163	2.67709
33554432	492.02380	9.74561
Siegrune		
2097152	88.94593	1.85451
4194304	188.71978	3.41997
8388608	409.09711	36.13593
16777216	909.34783	81.67
33554432	1910.8261	160.60827
Rossweisse		
2097152	95.40269	1.09854
4194304	204.66360	4.59815
8388608	428.42470	3.35943
16777216	951.22738	77.4042
33554432	1998.72261	152.8972
Grimgerde		
2097152	24.88658	10.20334
4194304	44.55758	0.86754
8388608	96.29102	1.46595
16777216	212.82059	2.54191
33554432	443.86681	10.12

Execution time metrics for benchmark 0					
Algorithm: polyphase merge sort					

Also let us notice that the maximum throughput for the disks and the `/work` directories (measured) is of 20MB/s for siegrune, grimgerde and helmvige and of 40MB/s for rossweisse. Our conclusion is that helmvige and grimgerde are 4 times more powerful than siegrune and rossweisse for sorting! When we read the characteristic of disk manufacturer, upon reading and given in Table 2.2, nobody would think that.

There was desire for concluding that the cluster was homogeneous. But it is not true. The technical explanation is due to the fact that rossweisse and siegrune are charged, at the time of the experiments, whereas helmvige and grimgerde are dedicated to us. We thus configured the vector of performance of the algorithm with the values {1, 1, 4, 4} and we guarantee until the end of our measurements that the machines remain in the same operating conditions.

Let us examine now the parallel case and discuss the performance of the parallel external algorithm listed in Table 2.4. With the cluster such as it is described in Table 2.2 and with a vector of performance containing only values 1, we can obtain, with Fast-Ethernet, results worse than the sequential case: we obtain an execution time of 133, 6 seconds!

Table 2.4: External parallel sorting on cluster (see Table 2.2), message size: 32Kbyte, 15 intermediate files, 30 experiments

Size	MET (s)	Stand. Dev.	Mean	Max	S(max)
Performance: {1, 1, 1, 1}; Fast-Ethernet					
16777216	303.94	9.173	4193043.8	4204494	1.00273
Performance: {1, 1, 4, 4}; Fast-Ethernet					
16777220	155.41	3.645	6816502.4	7342910	1.094
Performance: {1, 1, 4, 4}; Myrinet					
16777220	155.43	3.465	6293368.5	7341545	1.093
Execution time metrics for benchmark 0					
Algorithm: external PSRS					

On the other hand, with Fast-Ethernet always but with packets of data of 8K we obtain a sorting of 2097152 integers in 32.6 seconds. Now let us examine in Table 2.4 the results of experiments for sorting on disks 2^{24} = 16777216 integers on 4 processors while keeping a vector of performance not containing the values 1 but filled with the values {1, 1, 4, 4}.

Observe first the case not very favorable where the interconnected network is Fast-Ethernet (and not Myrinet).

The six columns in Table 2.4 correspond to (from left to right): problem size in number of integers, mean execution time (MET) expressed in seconds, standard deviation of the execution time, average of the size of all the final partitions (the optimal size is 4194304 for the homogeneous case), the size of the maximum partition, the "sublist expansion metric" which is the ratio between the values of column 5 and 4194304 (for the case where the vector of performance contains only values 1). Benchmark 0 corresponds to a pseudo-random generator.

We notice that the "sublist expansion metric" is in practice very close to the optimal 1. Thus the load is completely mastered. However, we find that the execution time is worse than the sequential time (235s) to sort the same quantity of information on Helmvige. It is however better than the sequential time of execution on Siegrune (909s). In this case the gain with 4 processors is 3.

The lowest common multiple of {1, 1, 4, 4} being 4, we can choose size 16777220 as the input problem size for the two last lines in Table 2.4. These two lines relate to an experimentation under Fast-Ethernet and an experimentation under Myrinet.

The optimal size for the two slowest processors (Siegrune and Rossweisse) is 1677722 whereas the optimal one for the two fastest processors (helmvige and grimgerde) is 6710888 integers. In Table 2.4, the "Mean" column of the non-homogeneous cases gives the average size of the data treated by the two fastest processors, the column "Max" gives the largest size of the data treated on the two fastest processors and the "S(Max)" column gives the "sublist expansion metric" for the two fastest processors.

Again, we note a value close to the optimal value which is 1. Compared to the most favorable sequential execution time (212s) we obtain a gain of 1.37; compared to the most unfavorable sequential execution time (951s) we obtain a gain of 6.13 by using 4 processors.

We note that a gain larger than the number of processors is possible! There is nothing mysterious in that! We obtain a gain from almost 2 (1.96) compared to the homogeneous configuration (when the vector of performance contains only values equal to 1)... that validates our approach.

Lastly, we note that the execution with Myrinet as support of communication does not improve the performances of the

execution time. The explanation holds due to the fact that the application communicates a few (but well): we have only one communication step. Moreover, the execution time takes into account neither the phase of distribution (the inputs are randomly generated on one host) nor the data-gathering phase (if we require it).

2.5.7 Parallel Sample Sort (PSS) revisited

2.5.7.1 Introduction

The key to the success in sorting is dependent of the splitters that must partition the initial bucket into roughly equal sizes. As noted in [VIT 94a], "It seems difficult to find $S = \Theta(m)$ splitters using $\Theta(n/D)$ I/Os[8] (the formation of the buckets must be done akin to this bound to guarantee an optimal algorithm) and guarantee that the bucket sizes are within a constant factor of one another".

The Parallel Sample Sort (PSS) algorithm [HUA 83] and its improvement [LI 94] does not sort the portions first but it uses *oversampling* to select pivots. It picks $p - 1$ pivots by randomly choosing $p * s$ candidates from the entire input data, where s is the oversampling ratio, and then selecting $p - 1$ pivots from the sorted candidates. Intuitively, a larger oversampling ratio results in better load balancing but increases the cost of selecting pivots.

We propose the following framework for external sorting which is based on PSS:

1. Pick pivots;
2. Broadcast pivot to a master node that sort them; keep $p - 1$ pivots and broadcast them to each node;
3. Each node partitions its input according to the pivots; broadcast the partitions;
4. Sort the received partitions (in our case we use again polyphase merge sort).

It can be shown [LI 94] that for an unsorted list of size n, $(pk - 1)$ pivots (with $k \geq 2$) partition the list into $p * k$ sublists such that the size of the maximum sublist is less or equal to n/p with probability at least $1 - 2p(1 - (1/(2p)))^{pk}$.

In the case of an heterogeneous cluster (processors at different speeds) we simulate a p' machine with p' equals the sum of coefficients in the performance vector. We also set k with $61\log_2 p'$ to mimic the framework of Li and Sevcik [LI 94].

Example:

for perf= {1, 1, 4, 4}, $k = 3$ we find that the size of the maximum sublist is less or equal to n/p' with probability at least $1-2*10(1-(1/(2*10)))^{10*3*log_2 10}$ that is to say with probability $1 - 20(0.95)^{99.6578} = 1 - 0.12 = 88\%$.

Now, if we set $k = 6 * \log_2 p'$ we get a probability of 99.92739%. For an out-of-core point of view, the increase (sustained by $k = 3$ becomes $k = 6$) on the number of pivots is acceptable because the memory usage stays low!

Finally, let p' be the sum of values in the performance vector. Thus the total number of pivots selected in our implementation of external PSS is: $p' * 6 \log p'$. Note that this number is quite low for an out-of-core point of view, so it fits in main memory. Moreover, it is a divisor of p'. Note also that the more the cluster is unbalanced (for instance a processor is 1000 times more powerful than the others) the more the probability is high... that is to say, we will have more chance to get balanced sublists. The choice of p' is thus justified to capture the heterogeneity of the machine!

In the forthcoming experiments, we use a small cluster composed of one Pentium III (Katmai), 451 Mhz, cache: 512KB, RAM: 261668 kB and 3 Celerons (Mendocino), 400 Mhz, cache: 128MB, RAM: 64MB. Disks were FUJITSU MPD3064AT disks with 512KB of cache.

Tables 2.5, 2.6, 2.7 and 2.8 are divided into five columns. From left to right, we have the mean size of data in the last step of the algorithm (Mean), the standard deviation of the mean (SD), the ratio of the mean over the optimal size, the ratio of the mean over the standard deviation and, at least the maximal and minimal observed sizes over the 35 experiments.

Table 2.5: Heterogeneous Sample Sort (2MB of data, heterogeneous configuration of performance vector)

Mean	SD	Mean/opt	Mean/SD	Max, Min
PID0				
115632	10847	93.88%	9.38%	96144, 10847
PID1				
371959	15981	100.09%	4.29%	392898, 335504
PID2				
615038	20479	100.10%	3.33%	652183, 571873

PID3				
986599	20970	100.36%	2.12%	1033844, 947255

Table 2.6: Heterogeneous Sample Sort (16MB of data, heterogeneous configuration of performance vector)

Mean	SD	Mean/opt	Mean/SD	Max, Min
PID0				
930196	87822	94.62%	9.44%	1108952, 759477
PID1				
2935879	157911	99.51%	5.38%	3364108, 2616418
PID2				
4974058	140542	101.2%	2.82%	5379087, 4709106
PID3				
7871546	211648	100.36%	2.69%	8153092, 7436021

Table 2.7: Heterogeneous Sample Sort (16MB of data, homogeneous configuration of performance vector)

Mean	SD	Mean/opt	Mean/SD	Max, Min
PID0				
3918862	584064	93,78%	14,90%	5744959, 3045409
PID1				
4150519	625341	99,34%	15,06%	6083675, 2942227
PID2				
3935862	579492	94,2%	14,72%	5423733, 2755896
PID3				
4706443	505979	112,65%	10,75%	5719919, 3699471

Table 2.8: Heterogeneous Sample Sort (2MB of data, homogeneous configuration of performance vector)

Mean	SD	Mean/opt	Mean/SD	Max, Min
PID0				
472349	59876	90.45%	12.67%	629023, 377398
PID1				
526403	52376	100.79%	9.95%	604596, 434675
PID2				
525042	62448	100.53%	11.89%	654503, 384518
PID3				
564548	47235	108.10%	8.36%	645109, 452583

2.5.7.2 Heterogeneous Sample Sort

We set the performance vector to {1, 3, 5, 8} and we observe the load balancing factor (in the "Mean/opt" column). A first result is presented in Table 2.5. We sort $n = 2088960$ integers and we use our benchmark numbered 0 (random generated data using the linear congruential generator $x_{k+1} = ax_k (\mathrm{mod}\ 2^{46})$).

After that, we set again the performance vector to {1, 3, 5, 8} and we observe the load balancing factor (in the "Mean/opt" column) in Table 2.6. Here we sort $n = 16711680$ integers.

When we compare the results about the load expansion metric in Tables 2.5 and 2.6 we observe a very good metric. The choice made for the number of pivots is appropriate.

2.5.7.3 Sample Sort with a performance vector configuration as an homogeneous cluster

We configure now our algorithm with the following setting for the performance vector: {1, 1, 1, 1}.

A first result is presented in Table 2.7. We sort n = 16711680 integers (the optimal amount of data per processor is 4177920 integers). We start 35 experiments and we observe a mean execution time of 82.15 seconds (the standard deviation is 6.75 seconds).

A second result is presented in Table 2.8. Here, we sort n = 2088960 integers (the optimal amount of data per processor is 522240 integers). We start 35 experiments and we observe a mean execution time of 6.25 seconds (the standard deviation is 0.56 seconds).

Again, the results about the load expansion metric are good. All the results validate the approach both for the heterogeneous case and for the homogeneous case. So, the external parallel sample sort algorithm developed in this section is of general use.

Our last remark concerns the way we fill the performance vector. We have said previously that a vector filled with the same value (1) represents the "homogeneous case". If we set entirely the vector with value 10 we also modelize the "homogeneous case" but if we run the program according to this setting we will generate more pivots!

2.5.8 Parallel sorting by overpartitioning revisited

2.5.8.1 Introduction

Li and Sevcik in [LI 94] proposed an algorithm for in-core sorting on homogeneous platforms with no sequential sort in the beginning. The choice and the number of pivots is carried out according to the discussion in the previous section: for an unsorted list of size n, $(pk - 1)$ pivots (with $k \geq 2$) partition the list into $p * k$ sublists such that the size of the maximum sublist is less or equal to n/p with a probability of at least $1 - 2p(1 - (1/(2p)))^{pk}$.

The algorithm presented in [LI 94] for sorting on homogeneous platforms with the overnartitioning technique is as follows:

Algorithm 2 (PSOP [LI 94])

Step 1 *initially, processor i has l_i, a portion of size n/p of the unsorted list l.*

Step 2 (selecting pivots) *a sample of p.k.s candidates is randomly picked from the list, where s is the oversampling ratio and k the over partitioning ratio. Each processor picks s.k candidates and passes them to a designated processor. These candidates are sorted and then $p.k - 1$ pivots are selected by taking (in a 'regular way') s^{th}, $2.s^{th}$, ..., $(pk - 1)^{th}$ candidates from the sample. The selected pivots d_1, d_2, ..., d_{pk-1} are made available to all the processors.*

Step 3 (partitioning) *since the pivots have been sorted, each processor performs binary partitioning on its local portion. Processor i decomposes l_i according to the pivots. It produces pk sublists per processor denoted l_{ij} where i, j stands for two consecutive pivots (except for the initial an final case). A sublist S_j is the union of l_{ij} with i ranging over all processors. There are pk sublists.*

Step 4 (building a task queue and sorting sublists) *Let $T(S_j)$ denotes the task of sorting S_j. The size of each sublist can be computed:*

$$|S_j| = \sum_{i=1}^{p} |l_{ij}|$$

Also the starting position of sublist S_j in the final sorted array can be calculated:

$$\sigma_j = 1 + \sum_{h=1}^{j-1} |S_h|$$

A task queue is built with the tasks ordered from the largest sublists size to the smallest. Each processor repeatedly takes one task $T(S_j)$ at a time from the queue. It processes the task by (a) copying the p parts of the sublist into the final array at position σ_j to $\sigma_j + |S_j| - 1$, and (b) applying a sequential sort to the elements in that range. The process continues until the task queue is empty.

2.5.8.2 The heterogeneous case

The main difference in the heterogeneous case is in the way we manage partitions and in the way we select pivots. First, the number of candidates is calculated according to $4 * p' * p' * \log_2(p')$ where p' is the sum of the values stored in the performance vector. After a sorting stage, we keep $4 * p' * \log_2(p') - 1$ pivots among the candidates. Note that this number is independent of the problem size and also that if p' grows (the cluster is more "unbalanced"), the number of pivots grows and

we amortize the risk of unbalanced partitions.

Second, Step 4 of Algorithm 2 is modified as follows: the partition sizes of task T_j, ($1 \leq j \leq$ number of partitions) are broadcast to processors and sorted. In order to decide if processor i keeps or rejects the task of sorting partition T_j, processor i computes $T_j.\ size$ divided by its performance where $T_j.size$ represents the number of elements in partition T_j. The ratio of $T_j.size$ and the performance gives an estimate of the "execution time of task T_j". We allocate task T_j on processor with the smallest execution time. A special protocol is also deployed in case of a draw but we ignore such details here. We also keep, when we visit task T_j in order to decide which processor will execute it, the sum of execution time of all previous tasks T_k, ($1 \leq k < j$) that has been allocated to processor i.

2.5.8.3 Experiments

We set the performance vector to {8, 5, 3, 1}. Tables 2.9 and 2.10 propose two experiments for input sizes of 1973785 and 16777215 integers stored initially on disks of a cluster of four processors. We notice that the load expansion metric (columns Mean/opt) are very good as well as the standard deviation of the observed values (column SD). The maximal and minimal values observed on processors are also good. We conclude that the number of pivots ($4 * 17 * log_2(17) - 1 = 271$ is very low comparing to the input size) is good enough to ensure a quasi-perfect load balance of the work. As a consequence, the overhead due to the processing of the partitions in the last step of the algorithm is kept low because we have fewer data to manage.

Table 2.9: Heterogeneous PSOP (1913785 integers, heterogeneous configuration of performance vector)

Mean	SD	Mean/opt	Max, Min
PID0			
929386	229	100.059%	929858, 929018
PID1			
580687	225	100.027%	581129, 580170
PID2			
347791	143	99.85%	348123, 347339
PID3			
115920	184	99.84%	116121, 115202

Table 2.10: Heterogeneous PSOP (16777215 integers, heterogeneous configuration of performance vector)

Mean	SD	Mean/opt	Max, Min
PID0			
7898307	1690	100.04%	7901360. 7895160
PID1			
4936858	1621	100.05%	4939629, 4933891
PID2			
2956149	1414	99.84%	2959340, 2953082
PID3			
985900	1115	99.89%	987923, 983598

[8] m and D are defined in the introduction to the chapter.

2.6 Conclusion

In this chapter we have presented algorithms to tacke the external sorting problem when processors in the cluster are heterogeneous. The heterogeneous notion, here, means that the performance of each node is related by constants.

The experiments concern both the acceleration and the load balance of each node. The goal is to bound the work done by each node by a constant closed to the number defining the initial amount of data stored on local disks: in an ideal way, if a node has initially X megabytes of data stored on its local disk, then we would want the node not to deal, during the execution of the external sorting algorithm, with more than X megabytes of data.

The results of experiments consolidate the idea that the algorithms and the implementations that we propose are serious candidates when the communication network is not fast and when the cluster is heterogeneous. So, the external sorting algorithms have good theoretical and practical properties. Table 2.11 summarizes the main properties of the different algorithms presented in this chapter.

Table 2.11: Summary of main properties of algorithms

Criteria	H-PSS	H-PSRS	H-PSOP
Number of candidates	$6 * p' * log_2(p')$	$p' * (p - 1)$	$4 * p' * p' * \log_2(p')$
Number of pivots	$P - 1$	$P - 1$	$4 * p' * \log_2(p') - 1$
Initial sort	No	Yes	No
Load balance (theory)	Algorithm manages partitions of size n/p' with a probability which is a function of the number of candidates.	No processor has more that two times its initial load	Algorithm manages partitions of size n/p' with a probability which is a function of the number of candidates.
Load balance (measured)	±15% of the optimal and in the worst case and on one processor.	±0.1% of the optimal value	±0.01% of the optimal value
Number of files created / proc	$15 + P + 1$	15 + P + 1	$15 + 4 * p' * \log_2(p')$
Sensitivity to duplicates	?	No, until a bound of n/p' duplicates	?
Message sizes	32KB	32KB	32KB
Allocated memory	$8K * sizeof(int) + 6 * p' * \log_2(p')$	$8K * sizeof(int) + p' * (p - 1)$	$8K * sizeof(int) + \mathcal{O}(p')$

In this table, H-PSS means "Heterogeneous Parallel Sample Sort". H-PSRS means "Heterogeneous Parallel Sorting by Regular Sampling" and H-PSOP means "Heterogeneous Parallel Sorting by OverPartitioning".

The reader should notice in particular that the memory (RAM) usage is very low as well as the number of files used in the implementations. In Table 2.11, constant 15 is the number of temporary files used by the polyphase merge sort we have employed, p' is the sum of values stored in the performance vector, P is the number of processors and n is the input size.

2.7 Bibliography

[ABE 99] Abello J., Vitter J. S., Eds., *External Memory Algorithms and Visualization*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society Press, Providence, RI, 1999.

[AGG 88] Aggarwal A., Vitter J. S., *"The Input/Output Complexity of Sorting and Related Problems"*, *Communications of the ACM*, vol. 31, num. 9, p. 1116-1127, September 1988.

[AKL 85] Akl S., *Parallel Sorting Algo. ithms*, Academic Press, 1985.

[A.S 95] A.Shirazi B., R.Hurson A., M.Kavi K., *Scheduling and Load Balancing in Parallel and Distributed Systems*, IEEE CS press, 1995.

[BAT 68] Batcher K. E., *"Sorting Networks and their Applications"*, *Proceedings of AFIPS Spring Joint Computer Conference*, p. 307-314, 1968.

[BLA 93] Blackston D. T., Ranade A., *"SnakeSort: A Family of Simple Optimal Randomized Sorting Algorithms"*, Hariri, SALIM; Berra P. B., Ed., *Proceedings of the 1993 International Conference on Parallel Processing. Volume 3: Algorithms and Applications*, Syracuse, NY, CRC Press, p. 201-204, August 1993.

[BLE 91] Blelloch G., Leiserson C., Maggs B., *"A Comparison of Sorting Algorithms for the Connection Machine CM-2"*, *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, July 1991.

[BOR 82] Borodin A., Hopcroft J. E., *"Routing, Merging, and Sorting on Parallel Models of Computation"*, *ACM Symposium on Theory of Computing (STOC '82)*, Baltimore, USA, ACM Press, p. 338-344, May 1982.

[CÉR 00] Cérin C., Gaudiot J.-L., *"Parallel Sorting Algorithms with Sampling Techniques on Clusters with Processors Running at different Speeds"*, *HiPC'2000. 7th International Conference on High Performance Computing. Bangalore, India*, Lecture Notes in Computer Science, Springer-Verlag, 17-20December 2000.

[CÉR 02] Cérin C., *"An Out-of-Core Sorting Algorithm for Clusters With Processors at Different Speed"*, *16th International Parallel and Distributed Processing Symposium (IPDPS), Ft Lauderdale, Florida, USA*, Page Available on CDROM from IEEE Computer Society, 2002.

[COL 88] Cole R., *"Parallel Merge Sort"*, *SIAM Journal of Computer*, vol. 17, p. 770-785, aug. 1988.

[COR 97] Cormen T. H., Hirschl M., *"Early experiences in evaluating the parallel disk model with the ViC* implementation"*, *Parallel Computing*, vol. 23, num. 4-5, p. 571-600, May 1997.

[DEW 91] Dewitt D. J., Naughton J. F., Schneider D. A., *"Parallel Sorting on a Shared-Nothing Architecture using Probabilistic Splitting"*, *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, p. 280-291, December 1991.

[HEL 96] Helman D. R., Jájá J., Bader D. A., A New Deterministic Parallel Sorting Algorithm With an Experimental Evaluation, Technical Report num. CS-TR-3670 and UMIACS-TR-96-54, Institute for Advanced Computer Studies, University of Maryland, College Park, MD, August 1996.

[HUA 83] Huang J. S., Chow Y. C., *"Parallel Sorting and Data Partitioning by Sampling"*, *IEEE Computer Society's Seventh International Computer Software & Applications Conference (COMPSAC'83)*, p. 627-631, November 1983.

[KIM 86] Kim M. Y., *"Synchronized Disk Interleaving"*, *IEEE Transactions on Computers, Vol. C-35*, num. 11, November 1986.

[KNU 98] Knuth D. E., *Sorting and Searching, vol. 3 of The Art of Computer Programming*, Addison-Wesley, Reading, MA, USA, second edition, 1998.

[LEI 84] Leighton T., *"Tight Bounds on the Complexity of Parallel Sorting"*, *ACM Symposium on Theory of Computing (STOC '84)*, Baltimore, USA, ACM Press, p. 71-80, April 1984.

[LI 93] Li X., Lu P., Schaeffer J., Shillington J., Wong P. S., Shi H., *"On the Versatility of Parallel Sorting by Regular*

Sampling", *Parallel Computing*, vol. 19, p. 1079-1103, October 1993.

[LI 94] Li H., Sevcik K. C., *"Parallel Sorting by Overpartitioning"*, *Proceedings of the 6th Annual Symposium on Parallel Algorithms and Architectures*, New York, NY, USA, ACM Press, p. 46-56, June 1994.

[MAY 00] May J. M., *Parallel I/O for High Performance Computing*, Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 2000.

[NAS 82] Nassimi D., Sahni S., *"Parallel permutation and sorting algorithms and a new generalized connection network"*, *J. ACM*, num. 29, p. 642-667, 1982.

[NOD 95] Nodine M. H., Vitter J. S., *"Greed Sort: Optimal Deterministic Sorting on Parallel Disks"*, *Journal of the ACM*, vol. 42, num. 4, p. 919-933, July 1995.

[PEA 99] Pearson M. D., Fast Out-of-Core Sorting on Parallel Disk Systems, Report num. PCS-TR99-351, Dept. of Computer Science, Dartmouth College, Hanover, NH, June 1999.

[PLA 89] Plaxton C. G., Efficient Computation on Sparse Interconnection Networks, Report num. STAN-CS-89-1283, Department of Computer Science, Stanford University, September 1989.

[QUI 89] Quinn M., *"Analysis and Benchmarking of two Parallel Sorting Algorithms: Hyperquicksort and Quickmerge"*, *BIT*, vol. 29, num. 2, p. 239-250, 1989.

[RAJ 98] Rajasekaran *"A Framework for Simple Sorting Algorithms on Parallel Disk Systems (extended abstract)"*, *SPAA: Annual ACM Symposium on Parallel Algorithms and Architectures*, 1998.

[REI 83] Reif J. H., Valiant L. G., *"A Logarithmic Time Sort for Linear Size Networks"*, *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, Boston, Massachusetts, p. 10-16, 25-27 April 1983.

[REI 87] Reif J. H., Valiant L. G., *"A Logarithmic time Sort for Linear Size Networks"*, *Journal of the ACM*, vol. 34, num. 1, p. 60-76, January 1987.

[SAL 86] Salem K., Garcia-Molina H., *"Disk Striping"*, *Proceedings of the 2nd International Conference on Data Engineering*, ACM, p. 336-342, February 1986.

[SHI 92] Shi H., Schaeffer J., *"Parallel Sorting by Regular Sampling"*, *Journal of Parallel and Distributed Computing*, vol. 14, num. 4, p. 361-372, 1992.

[TRI 93] Tridgell A., Brent R., An Implementation of a General-purpose Parallel Sorting Algorithm, Report num. TR-CS-93-01, Computer Science Laboratory, Australian National University, February 1993.

[VIT 94a] Vitter J. S., Shriver E. A. M., *"Algorithms for Parallel Memory I: Two-level Memories"*, *Algorithmica*, vol. 12, num. 2/3, p. 110-147, August and September 1994.

[VIT 94b] Vitter J. S., Shriver E. A. M., *"Algorithms for Parallel Memory II: Hierarchical Multilevel Memories"*, *Algorithmica*, vol. 12, num. 2/3, p. 148-169, August and September 1994.

Part Two: Selected Readings

Chapter List

Chapter 3: A Sensitivity Study of Parallel I/O under PVFS-Lessons Learned Using a Parallel File System on PC Clusters

Jens Mache, Joshua Bower-Cooley, Robert Broadhurst, Jennifer Cranfill and Clark Kirkman, Lewis & Clark College, Portland, OR 97219, USA

3.1 Introduction

In order to compete with custom-made supercomputers, PC clusters have to provide not only fast computation and communication, but also high-performance disk access. Using the Parallel Virtual File System (PVFS) and a ray tracing application, we studied the sensitivity of parallel I/O performance, the overlapping of I/O nodes with compute nodes and the exploitation of local data. Our results show how parallel I/O throughput not only depends on hardware (disk speed and network bandwidth), but also configuration and programming choice.

With the recent popularity of cluster computing, a parallel file system is needed to provide high-performance disk access to data that is processed at increasing speeds. Many applications need to transfer large amounts of data to and from secondary storage. and I/O performance can play a critical role in the overall completion time of these applications. In order to compete with custom-made supercomputers, clusters have to provide not only fast computation and communication, but also fast parallel I/O [SCH 99, COR 99, SCH 00].

Efficient parallel I/O is considered "tricky" [SAP 99]. While previous work focused on custom-made supercomputers, we set out to measure the parallel I/O performance of PC clusters. In this paper, we explore the effectiveness of PVFS in increasing the performance of a ray tracing application. We study the sensitivity of parallel I/O performance, the overlapping of I/O nodes with compute nodes and the exploitation of local data.

The remainder of this chapter is organized as follows: we briefly discuss related work in Section 2. Section 3 gives background on parallel I/O and parallel file systems. Section 4 contains our performance evaluation. Our results are summarized in Section 5.

3.2 Related work

We briefly discuss related work in the area of parallel file systems and performance evaluation of parallel file systems.

Regarding parallel file systems, there are commercial file systems such as GPFS [BAR] for the IBM SP or Intel's PFS [GAR 98] for the Intel Paragon and ASCI Red. Unfortunately, none of these run on PC clusters. Another group of parallel file systems are research projects such as xFS [AND 96], River [ARP 99], PIOUS [MOY 96], PPFS [HUB 95], VIP-FS [HAR 95], Galley [NIE 97], Armada [OLD 01] and GFS [PRE 99]. These research prototypes seem not to be intended (or ready) for everyday use by general users. Currently, the most mature candidates for PC clusters seem to be Sun's PFS [Sun] for Solaris and the Parallel Virtual File System (PVFS) [Lig 96, Lig 99, CAR 00] from Clemson University for Linux.

Similarly, performance evaluation of parallel file systems seem to have focused on custom-made supercomputers. Koniges et al. [JON 00, PRO 00] studied GPFS on the IBM SP. Thakur et al. [THA 96] measured the performance of PIOFS on the IBM SP and Intel's PFS on the Intel Paragon. Bordawekar et al. studied the HP Exemplar file system, as well as the Concurrent File System (CFS) on the Intel Touchstone Delta. Kwan and Reed measured the performance of the CM-5 Scalable File System. Feitelson et al. studied the performance of the Vesta file system. Nieuwejaar and Kotz presented performance results for the Galley parallel file system. Several researchers have measured the performance of the Concurrent File System (CFS) on the Intel iPSC/2 and iPSC/860 hypercubes.

3.3 Background

This section provides background information on parallel I/O and parallel file systems in general, as well as PVFS in particular.

3.3.1 Parallel I/O and parallel file systems

The gap between computational power and I/O performance is widening. In recent years, CPU speed and network bandwidth have increased by about 60 percent annually, while disk access time has only improved by about 40 percent annually [JAI 96, MAY 01].

The idea of *parallel I/O* is to increase both storage capacity and I/O performance by storing data across multiple disks. A *parallel file system* transparently stripes data across multiple disks and I/O nodes. It provides a global name space which results in simplified file management and flexible access to files (e.g. independent of the number of compute nodes of a parallel application).

3.3.2 PVFS

At this time, the Parallel Virtual File System (PVFS) [Lig 96, Lig 99, CAR 00] from Clemson University seems to be the most mature parallel file system for Linux clusters. It is a user-level, client-server system utilizing TCP and the existing file system on each I/O node. File data is striped across I/O nodes according to user specification. Each I/O node runs a daemon that controls reading and writing for that node. A single manager daemon stores meta-data and controls file operations including open, close, and remove commands. The I/O daemons handle all data transfers; contacting the manager is only necessary to open and close the file. This allows compute nodes to transfer data directly to (or from) I/O nodes.

When creating a file, the user has control over the *physical* striping of data across I/O nodes as defined by a set of variables. It is possible to explicitly specify the number of I/O nodes (*pcount*), the physical stripe size (*ssize*) of the data, and a base node that determines the location of the beginning of the file. These variables determine the layout of data on the I/O nodes.

When writing to or reading from a file, applications can specify a *logical* partition of data that is independent of the physical data distribution. This logical partition defines the *file view*, controlling the part of the file seen by each compute node. A logical partition is specified by the size of each noncontiguous data block seen by the compute node (*gsize*), the location of the first block relative to the beginning of the file (*offset*) and the distance between two blocks seen by the compute node (*stride*).

For the example shown in Figure 3.1, there are two compute nodes sending a total of 6 logical blocks of data to two I/O nodes. The logical offset for compute node 1 is 1 block, the logical group size is 1 block, the logical stride is 2 blocks, and the physical stripe size is half of a block.

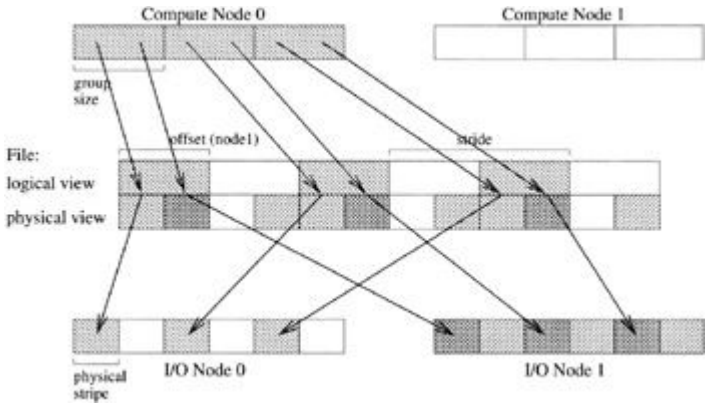


Figure 3.1: Two compute nodes sending a total of 6 logical blocks of data to two I/O nodes

3.4 Performance evaluation

This section describes our experiments. Our goal is to measure the sensitivity of parallel I/O performance of PC clusters running PVFS. We first describe our cluster and application programs.

3.4.1 Experimental configuration

Our cluster consisted of up to 12 Pentium III 450 MHz nodes with 128 MB of memory (PC100 SDRAM). Our interconnect was either switched Fast Ethernet or switched Gigabit Ethernet [MAC 99]. We were running Red Hat Linux 6.0 (kernel 2.2.5-15) and had MPICH 1.1.2 and PVFS 1.4.3 installed.

To measure parallel I/O performance under PVFS, we used two application programs. The first one is a "concurrent read/ write" test program. This MPI program writes and then reads blocks of integer data to and from a PVFS file. Each compute node issues six write calls of 16 MB each, creating a file that is (96* *number_of_compute_nodes*) MB in size. This amount of data is sufficient to effectively saturate all buffers used in local write processes on our cluster. The file is also opened and closed for each write to ensure that the process has been completed. Reads are accomplished in a similar fashion, with each compute node reading back the same data it has written. Each compute node records an elapsed wall clock time for every read and write. To obtain a throughput figure, the slowest time on the slowest node is used.

Our second application program is a ray tracer [MAC 00]. Ray tracing is a technique for rendering a scene composed of mathematically defined objects by following the set of rays that emanate from the observer and pass through the various pixels of the projection screen. Instead of displaying the calculated frames, we save the frames to disk. In our experiments, the ray tracer computes 10 frames of a simple scene at 840x840 pixels each, resulting in 27 MB of data to be saved across the specified number of I/O nodes. We report the average completion time of 40 executions.

3.4.2 Sensitivity to hardware

In the first set of experiments, we study the impact of hardware, namely disk speed and network bandwidth, on parallel I/O throughput. We had two sets of disks, IDE drives (Maxtor 90913D4) and SCSI drives (Seagate Barracuda ST39175LW). Using the Bonnie Disk Benchmark [BRA], we measured a disk read performance of 3-7 MBytes/sec for IDE and 15-17 MBytes/sec for SCSI, and a disk write performance of 4-6 MBytes/sec for IDE and 16-18 MBytes/sec for SCSI. Switching from IDE to SCSI disks, we thus expect a parallel I/O performance increase by a factor of 3 to 5.

However, to achieve full I/O performance from a remote node, network performance should not be smaller than disk performance. Using the ttcp benchmark [Tes], we measured a network performance of approximately 10 MBytes/sec for Fast Ethernet and 35 MBytes/sec for Gigabit Ethernet (with the drivers of the 2.2.x kernel and without optimizations). Thus, to utilize the full potential of our SCSI disks from remote nodes, a faster interconnect than Fast Ethernet is necessary.

Running our concurrent read/write test program, we measured a parallel I/O performance increase of a factor of 3 to 5, as expected, when switching from IDE disks and Fast Ethernet to SCSI disks and Gigabit Ethernet. For example, the read throughput of 3 compute nodes was 13.8 MBytes/sec from 3 IDE disks and 54.2 MBytes/sec from 3 SCSI disks. More I/O measurements are shown in Figures 3.2 and 3.3 (which are explained in more detail below).

3.4.3 Sensitivity to configuration

In the second set of experiments, we study the impact of configuration choices, namely number of I/O nodes, number of compute nodes and overlapping I/O nodes with compute nodes.

3.4.3.1 Number of I/O nodes

Parallel I/O performance is expected to scale linearly with the number of I/O nodes. Running our concurrent read/ write test program, measurements for IDE disks and Fast Ethernet are shown in Figure 3.2, and measurements for SCSI disks and Gigabit Ethernet are shown in Figure 3.3. Different lines report parallel I/O throughput for different number of I/O nodes. On the x-axis, number of compute nodes are varied (which we discuss below). If we look at the measurements for one I/O node and one compute node, two I/O nodes and two compute nodes, and so on, we see a roughly linear speed-up in all four graphs, as expected. The slope of a regression line is approximately 4.6 MBytes/sec for reading from IDE, 4.2 MBytes/sec for writing to IDE, 17 MBytes/sec for reading from SCSI and 20 MBytes/sec for writing to SCSI.

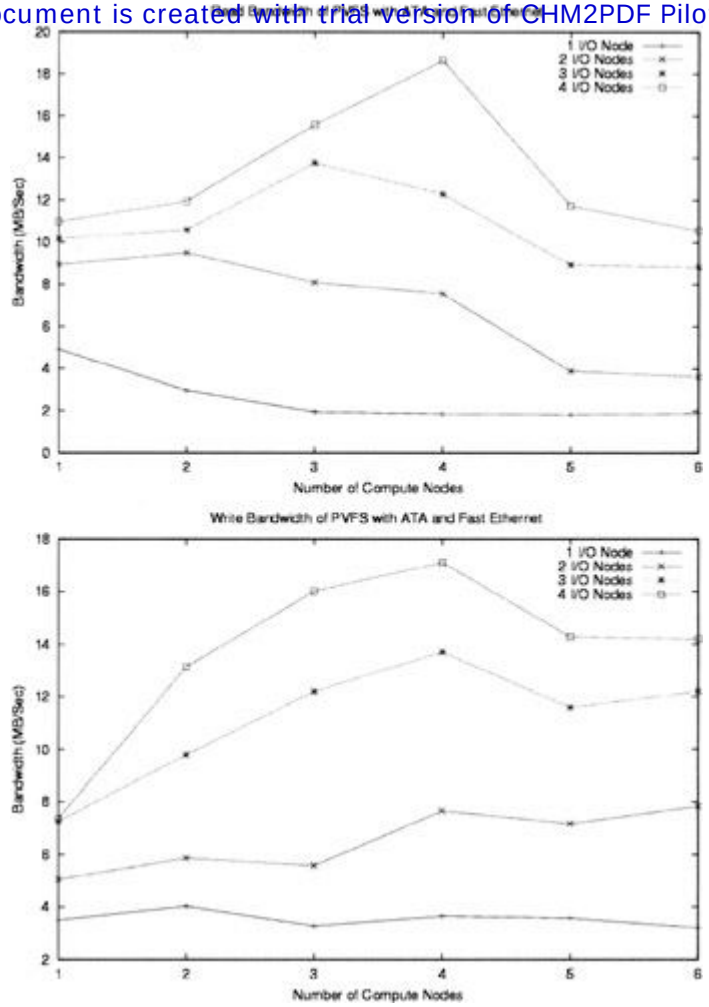


Figure 3.2: Read performance (top) and write performance (bottom) for IDE disks and Fast Ethernet

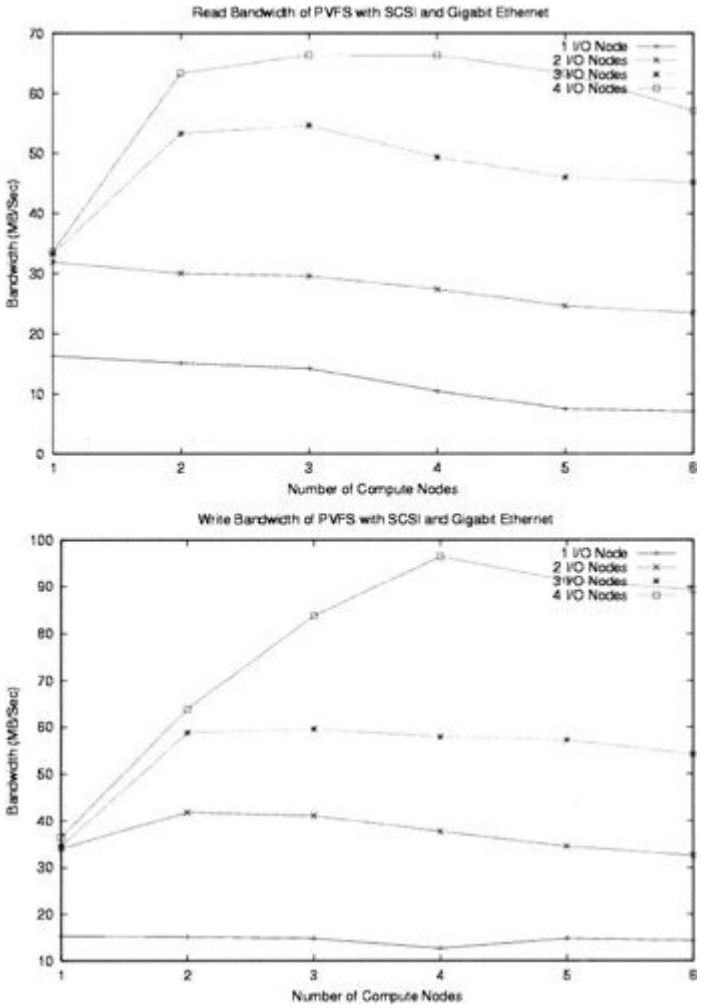


Figure 3.3: Read performance (top) and write performance (bottom) for SCSI disks and Gigabit Ethernet

3.4.3.2 Number of compute nodes

Parallel I/O performance is expected to be independent of number of compute nodes. (With the exception of too few compute nodes and network links to produce or consume data fast enough.)

However, Figures 3.2 and 3.3 show that parallel I/O throughput drops as more compute nodes access the disks concurrently. For example, when writing to 3 SCSI disks, we measured 60 MBytes/sec from 3 compute nodes, 58 MBytes/sec from 4 compute nodes, 57 MBytes/sec from 5 compute nodes and 54 MBytes/sec from 6 compute nodes.

This indicates a disk contention problem. The smaller the number of clients, the more effective server-side caching. Whereas future versions of PVFS may implement more sophisticated scheduling at the I/O daemons [CAR 00, HAD 00], application programmers could (1) limit the number of compute nodes that access disks concurrently or (2) consider I/O scheduling [JAI 97] or data block scheduling [NIT 95] at the application level.

3.4.3.3 Number of I/O vs. compute nodes if cluster size is fixed

If the total number of nodes is fixed and nodes do not double as I/O and compute nodes (see below), adding one I/O node takes away one compute node. To study this effect, we ran experiments with our ray tracing application that needs both I/O and compute performance. We measure overall completion time while varying the number of compute and I/O nodes on an 8 node cluster. Seven of the experiments involved distinct I/O and compute nodes, from 1 I/O node with 7 compute nodes to 7 I/O nodes receiving data from 1 compute node.

Completion time results with IDE disks and Fast Ethernet are shown in Figure 3.4. The 1&7 configuration (1 I/O node and 7 compute nodes) completed in 6.22 sec. For this particular application, completion time was lowest in the 2&6 configuration (5.03 sec).

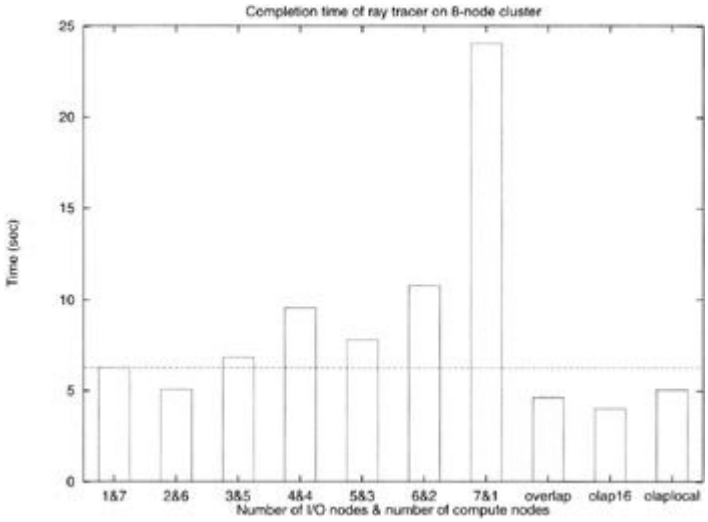


Figure 3.4: Completion time of ray-tracing application depending on number of I/O nodes and number of compute nodes

3.4.3.4 Overlapped I/O and compute nodes

While it is possible to configure PVFS with overlapped I/O and compute nodes, is it beneficial?

On one hand, I/O nodes need compute cycles to manage the data transfers and to calculate which data goes where. It is not uncommon to use 20% of CPU time. A compute node that doubles as I/O node cannot compute as fast as a "full-time" compute node. This could result in longer completion times of the application (especially if load balancing is an issue).

On the other hand, every cluster has a limited number of nodes, and dedicating some to I/O will reduce the number of possible compute nodes. It may be advantageous to use all nodes in a cluster for both computation and I/O.

Ray tracer measurements for three overlapped configurations are shown on the far right of Figure 3.4. Completion times for overlapped configurations are lower, as low as 3.98 seconds. (More details about the three different experiments are provided below).

3.4.4 Sensitivity to programming choices

In the last set of experiments, we study the impact of programming choices, namely logical file view, physical stripe size and exploiting local data.

3.4.4.1 Logical file view

With a logical file view, non-contiguous file regions can be accessed with a single I/O call and optimizations can be

performed. For the example shown in Figure 3.1, compute node 0 has to issue 1 instead of 3 write calls. In our experiments, using logical file view improved I/O throughput by up to a factor of 1.45. (We used a logical view for all performance numbers reported in this paper.)

3.4.4.2 Physical stripe size

We initially used a physical stripe size of 10 lines of the picture (32.81 KB) so that physical stripe size divides logical distribution size evenly. However, when we changed the physical stripe size to 16 KB, a multiple of the disk block size, completion time decreased from 4.61 seconds for the "overlap" experiment (see Figure 3.4) to 3.98 seconds for the "olap16" experiment. It seems to be more important to choose a physical stripe size that fits the disk rather than one that fits the logical data distribution.

3.4.4.3 Exploiting local data

While it is possible to store data exclusively on local disks (by setting an appropriate file view and physical stripe size for overlapped nodes), is it beneficial? Theoretically, network traffic [CHO 97] and disk contention could be avoided.

However, when using this technique with our ray tracer, completion time went up, from 4.61 seconds for the "overlap" experiment to 5.03 seconds for the "olaplocal" experiment, see Figure 3.4. An explanation is that because nodes do not always transfer data at the same time, (1) conflicts seem to not be a problem and (2) each node can finish its I/O faster if it transfers to all disks instead of to its local disk only.

3.5 Conclusions

High performance disk access is becoming a key topic in cluster computing. Although we were able to reduce the completion time of our ray tracing application by a factor of 1.563 by using PVFS, I/O throughput was sensitive to hardware, configuration and programming choices.

Our main conclusions are as follows:

- Parallel I/O throughput is sensitive to hardware. By using faster disks, I/O throughput increased as expected, as long as the interconnection network was not the limiting factor.
- Parallel I/O throughput is sensitive to configuration.
  - I/O performance is not only dependent on number of I/O nodes (as expected) but also on number of compute nodes (due to disk contention).
  - We achieved the best completion time for our ray tracing application using overlapped compute and I/O nodes.
- Parallel I/O throughput is sensitive to programming choices.
  - It is beneficial to set up a logical file view.
  - We achieved best performance using a physical stripe size that fit the disk rather than the logical data distribution.
  - For our ray tracing application, storing data exclusively on local disks resulted in poorer performance overall.

These results are highly valuable (1) to give performance recommendations for application development and (2) as a guide to I/O benchmarking (which will play an important role in compiling the new "clusters @ top500" ranking [Clu]).

3.6 Bibliography

[AND 96] Anderson T. E., Dahlin M. D., Neefe J. M., Patterson D. A., Roselli D. S., Wang R. Y., *"Serverless network file systems"*, *ACM Transactions on Computer Systems*, vol. 14, num. 1, p. 41-79, February 1996.

[ARP 99] Arpaci-Dusseau R. H., Anderson E., Treuhaft N., Culler D. E., Hellerstein J. M., Patterson D., Yellick K., *"Cluster I/O with River: Making the fast case common"*, *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems*, 1999.

[BAR] Barrios M., ET AL, *"GPFS: A Parallel File System"*, http://www.redbooks.com/ .

[BAR] Bray T., *"Bonnie file system benchmark"*, http://www.textuality.com/bonnie/ .

[CAR 00] Carns P. H., Ligon III W. B., Ross R. B., Thakur R., *"PVFS: A Parallel File System for Linux Clusters"*, *Proceedings of Extreme Linux*, 2000.

[CHO 97] Cho Y., Winslett M., Subramaniam M., Chen Y., Kuo S., Seamons K. E., *"Exploiting Local Data in Parallel Array I/O on a Practical Network of Workstations"*, *Proceedings of the 5th Workshop on I/O in Parallel and Distributed Systems*, SC '97, p. 1-13, 1997.

[Clu] CLUSTERS @ TOP500, http://clusters.top500.org .

[COR 99] Cortes T., *"Software RAID and Parallel Filesystems"*, Buyya R., Ed., *High Peformance Cluster Computing*, p. 463-496, Prentice Hall PTR, 1999.

[GAR 98] Garg S., *"TFLOPS PFS: Architecture and design of a highly efficient parallel file system"*, *Proceedings of Supercomputing '98*, 1998.

[HAD 00] Haddad I. F., *"PVFS: A Parallel Virtual File System for Linux Clusters"*, *Linux Journal*, p. 74-82, December 2000.

[HAR 95] Harry M., Del Rosario J. M., Choudhary A., *"VIP-FS: A Virtual, Parallel File System for High Performance Parallel and Distributed Computing"*, *Proceedings of the Ninth International Parallel Processing Symposium*, 1995.

[HUB 95] Huber J., Elford C. L., Reed D. A., Chien A. A., Blumenthal D. S., *"PPFS: A High Performance Portable Parallel File System"*, *Proceedings of the 9th ACM International Conference on Supercomputing*, 1995.

[JAI 96] Jain R., Werth J., Browne J. C., *"I/O In Parallel and Distributed Systems: An Introduction"*, Jain R., Werth J., Browne J. C., Eds., *Input/Output in Parallel and Distributed Computer Systems, vol. 362 of The Kluwer International Series in Engineering and Computer Science*, Chapter 1, p. 3-30, Kluwer Academic Publishers, 1996.

[JAI 97] Jain R., Somalwar K., Werth J., Browne J. C., *"Heuristics for Scheduling I/O Operations"*, *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, num. 3, p. 310-320, March 1997.

[JON 00] Jones T., Koniges A., Yates R. K., *"Performance of the IBM General Parallel File System"*, *Proceedings of IPDPS*, 2000.

[Lig 96] Ligon III W. B., Ross R. B., *"Implementation and Performance of a Parallel File System for High Performance Distributed Applications"*, *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing, IEEE Computer Society Press*, p. 471-480, August 1996.

[Lig 99] Ligon III W. B., Ross R. B., *"An Overview of the Parallel Virtual File System"*, *Proceedings of Extreme Linux*, 1999.

[MAC 99] Mache J., *"An Assessment of Gigabit Ethernet as Cluster Interconnect"*, *Proceedings of the First IEEE International Workshop on Cluster Computing*, December 1999.

[MAC 00] Mache J., Broadhurst R., Ely J., *"Ray Tracing on Cluster Computers"*, *Proceedings of CC-TEA*, 2000.

[MAY 01] May J. M., *Parallel I/O for High Performance Computing*, Morgan Kaufmann, 2001.

[MOY 96] Mover S. A., Sunderam V. S., *"Scalable concurrency control for parallel file systems"*, Jain R., Werth J., Browne J. C., Eds., *Input/Output in Parallel and Distributed Computer Systems, vol. 362 of The Kluwer International Series in Engineering and Computer Science*, Chapter 10, p. 225-243, Kluwer Academic Publishers, 1996.

[NIE 97] Nieuwejaar N., Kotz D., *"The Galley Parallel File System"*, *Parallel Computing*, vol. 23, num. 4, p. 447-476, North-Holland (Elsevier Scientific), June 1997.

[NIT 95] Nitzberg W. J. Collective Parallel I/O, PhD thesis, Department of Computer and Information Science, University of Oregon, December 1995.

[OLD 01] Oldfield R., Kotz D., *"Armada: A parallel file system for computational grids"*, *Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2001.

[PRE 99] Preslan K. W., Barry A. P., Brassow J. E., Erickson G. M., Nygaard E., Sabol C. J., Soltis S. R., Teigland D. C., O'keefe M. T., *"A 64-bit, Shared Disk File System for Linux"*, *Proceedings of the Seventh NASA Goddard Conference on Mass Storage Systems*, 1999.

[PRO 00] Prost J.-P., Treumann R., Blackmore R., Hartman C., Hedges R., Jia B., Koniges A., White A., *"Towards a High-Performance and Robust Implementation of MPI-IO on top of GPFS"*, *Proceedings of Euro-Par*, 2000.

[SAP 99] Saphir B., Bozeman P., Evard R., Beckman P., *"Production Linux Clusters: Architecture and System Software for Manageability and Multi-user Access"*, Tutorial at SC99, 1999.

[SCH 99] Schikuta E., Stokinger H., *"Parallel I/O for Clusters: Methodologies and Systems"*, Buyya R., Ed., *High Peformance Cluster Computing*, p. 439-462, Prentice Hall PTR, 1999.

[SCH 00] Schikuta E., Wanek H., *"Parallel I/O"*, Baker M., Ed., *Cluster Computing White Paper*, Chapter 7, December 2000.

[Sun] SUN PARALLEL FILE SYSTEM, http://www.sun.com/software/hpc/specifications.html .

[Tes] TEST TCP, ftp://ftp.arl.mil/pub/ttcp/ .

[THA 96] Thakur R., Gropp W., Lusk E., *"An Experimental Evaluation of the Parallel I/O Systems of the IBM SP and Intel Paragon Using a Production Application"*, *Proceedings of the 3rd International Conference of the Austrian Center for Parallel Computation*, 1996.

Chapter 4: The Parallel Effective I/O Bandwidth Benchmark-b_eff_io

Rolf Rabenseifner, High-Performance Computing Center (HLRS), University of Stuttgart, Allmandring 30, D-70550 Stuttgart, Germany
Alice E. Koniges, Lawrence Livermore National Laboratory, Livermore, CA 94550, USA
Jean-Pierre Prost, IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, USA
Richard Hedges, Lawrence Livermore National Laboratory, Livermore, CA 94550, USA

4.1 Introduction

The parallel effective I/O bandwidth benchmark (b_eff_io) is aimed at producing a characteristic average number of the I/O bandwidth achievable with parallel MPI-I/O applications exhibiting various access patterns and using various buffer lengths. It is designed so that 15 minutes should be sufficient for a first pass of all access patterns. First results of the b_eff_io benchmark are given for the IBM SP, Cray T3E, Hitachi SR 8000, and NEC SX-5 systems, and a discussion follows about problematic issues of our current approach.

We show how a redesign of our time-driven approach allows for rapid benchmarking of I/O bandwidth with various compute partition sizes. Next, we present how implementation specific file hints can be enabled selectively on a per access pattern basis, and we illustrate the benefit that hints can provide using the latest version of the IBM MPI-IO/GPFS prototype.

Crucial to the ultimate useful performance of a cluster computing environment is the seamless transfer of data between memory and a filesystem. Most cluster applications would benefit from a decent parallel filesystem that allows the transfer of data to occur using standard I/O calls such as those implemented in the MPI-2 standard MPI-I/O [MPI 97]. However, since there is a variety of data storage and access patterns that span the gamut of cluster applications, designing a benchmark to aid in the comparison of filesystem performance is a difficult task.

Indeed, many parallel I/O benchmarks and benchmarking studies characterize only the hardware and file system performance limits [DET 98, HAS 98, HO 99, JON 00, KOE 98]. Often, they focus on determining under which conditions the maximal file system performance can be reached on a specific platform. Such results can guide the user in choosing an optimal access pattern for a given machine and filesystem, but do not generally consider the needs of the application over the needs of the filesystem.

Other benchmarks, such as BTIO [CAR 92], are combining numerical kernels with MPI-I/O. The MPI-I/O benchmark described in [LAN 98] uses low-level benchmarks and kernel style benchmarks. This project emphasizes the development of a test suite for MPI-I/O together with I/O performance issues.

In this article, we describe the design and implementation of a parallel I/O benchmark useful for comparing filesystem performance on a variety of architectures, including, but not limited to cluster systems.

This benchmark, referred to as the parallel effective I/O bandwidth benchmark (b_eff_io in short), is aimed at:

a. getting detailed information about several access patterns and buffer lengths,

b. measuring a characteristic average number for the I/O bandwidth achievable with parallel MPI-I/O applications.

b_eff_io examines "first write", "rewrite", and "read" accesses, strided (individual and shared pointers) and segmented collective accesses to one shared file per application, as well as non-collective access to one file per process. The number of parallel accessing processes is also varied, and wellformed I/O is compared with non-wellformed I/O. On systems meeting the rule that the total memory can be written to disk in 10 minutes, the benchmark should not need more than 15 minutes for a first pass of all access patterns.

This article is structured as follows. In Section 4.2, we describe the basic design criteria that influenced our choice of appropriate I/O patterns. In Section 4.3, we give the specific definition of the benchmark. Results of the benchmark on a variety of platforms are described in Section 4.4. In Section 4.5, we discuss a few problematic issues related to the benchmark definition. In Section 4.6, we show how our time-driven approach can be applied to allow for rapid benchmarking of various compute partition sizes. In Section 4.7, we present how MPI-I/O file hints can be selectively enabled on a per access pattern basis in an attempt to improve the parallel I/O bandwidth achieved for that access pattern, and we illustrate this technique by describing our current experimentation using the latest IBM MPI-I/O prototype on an SP system. Finally, we mention future work and conclude the article.

4.2 Benchmark design considerations

We begin by considering the common I/O patterns of parallel applications. To standardize I/O requests, the MPI Forum introduced the MPI-I/O interface [MPI 97] that allows the user to make these requests using a convenient interface similar to standard MPI calls. Furthermore, MPI-I/O allows for optimization of accesses to the underlying filesystem (see for example [DIC 98, PRO 00, THA 99, THA1]), while retaining the uniformity of the basic MPI-I/O interface across platforms.

Based on this background, the parallel effective I/O bandwidth benchmark should measure different access patterns, report these detailed results, and should calculate an average I/O bandwidth value that characterizes the whole system. This goal is analogous to the Linpack value reported in TOP500 [TOP500] that characterizes the computational speed of a system, and also to the effective bandwidth benchmark (b_eff), that characterizes the communication network of a distributed system [RAB1, SOL 99, SOL1]. Indeed, the experience with the design of the b_eff benchmark is influential on this I/O benchmark design.

However, the major difference between b_eff and b_eff_io is the magnitude of the bandwidth. On well-balanced systems in high performance computing we expect an I/O bandwidth which allows for writing or reading the total memory is approximately 10 **minutes**. For the communication bandwidth, the b_eff benchmark shows that the total memory can be communicated in 3.2 **seconds** on a Cray T3E with 512 processors and in 13.6 seconds on a 24 processor Hitachi SR 8000. An I/O benchmark measures the bandwidth of data transfers between memory and disk. Such measurements are highly influenced by buffering mechanisms of the underlying I/O middleware and filesystem details, and high I/O bandwidth on disk requires, especially on striped filesystems, that a large amount of data be transferred between such buffers and disk. Therefore an I/O benchmark must ensure that a sufficient amount of data is transferred between disk and the application's memory. The communication benchmark b_eff can give detailed answers in about 2 minutes. Later we shall see that b_eff_io, our I/O counterpart, needs at least 15 minutes to get a first answer.

4.2.1 Multidimensional benchmarking space

Often, benchmark calculations sample only a small subspace of a multidimensional parameter space. One extreme example is to measure only one point, e.g., a communication bandwidth between two processors using a ping-pong communication pattern with 8 Mbyte messages, repeated 100 times. For I/O benchmarking, a huge number of parameters exist. We divide the parameters into 6 general categories. At the end of each category in the following list, a first hint about handling these aspects in b_eff_io is noted. The detailed definition of b_eff_io is given in section 4.3.

1. Application parameters are (a) the size of contiguous chunks in memory, (b) the size of contiguous chunks on disk, which may be different in the case of scatter/gather access patterns, (c) the number of such contiguous chunks that are accessed with each call to a read or write routine, (d) the file size, (e) the distribution scheme, e.g., segmented or long strides, short strides, random or regular, or separate files for each node, and (f) whether or not the chunk size and alignment are wellformed, e.g., a power of two or a multiple of the striping unit. For b_eff_io, 36 different patterns are used to cover most of these aspects.

2. Usage aspects are (a) how many processes are used and (b) how many parallel processors and threads are used for each process. To keep these aspects outside of the benchmark, b_eff_io is defined as a maximum over these aspects and one must report the usage parameters used to achieve this maximum.

3. The major programming interface parameter is specification of which I/O interface is used: Posix I/O buffered or raw, special filesystem I/O of the vendor's file system, or MPI-I/O. In this benchmark, we use only MPI-I/O, because it should be a portable interface of an optimal implementation on top of Posix I/O or the special filesystem I/O.

4. MPI-I/O defines the following orthogonal aspects: (a) access methods, i.e., first writing of a file, rewriting, or reading, (b) positioning method, i.e., explicit offsets, individual or shared file pointers, (c) coordination, i.e., accessing the file collectively by a group of processes or noncollectively, (d) synchronism, i.e., accessing the file in a blocking mode or in a nonblocking mode. Additional aspects are: (e) whether or not the files are open *unique*, i.e., the files will not be concurrently opened by other open calls, and (f) which consistency is chosen for conflicting accesses, i.e., whether or not atomic mode is set. For b_eff_io there is no overlap of I/O and computation, therefore only blocking calls are used. Because there should not be a significant difference between the efficiency of using explicit offsets or individual file pointers, only the individual and shared file pointers are benchmarked. With regard to (e) and (f), *unique* and *nonatomic* are used.

5. Filesystem parameters are (a) which filesystem is used, (b) how many nodes or processors are used as I/O servers, (c) how much memory is used as bufferspace on each application node, (d) the disk block size, (e) the striping unit size, and (f) the number of parallel striping devices that are used. These aspects are also outside the scope of b_eff_io. The chosen filesystem, its parameters, and any usage of non-default parameters must be reported.

6. Additional benchmarking aspects are (a) repetition factors, and (b) how to calculate b_eff_io, based on a subspace of the parameter space defined above using maximum, average, weighted average, or logarithmic averages.

To reduce benchmarking time to an acceptable amount, one can normally only measure I/O performance at a few grid points of a 1-5 dimensional subspace. To analyze more than 5 aspects, usually more than one subspace is examined. Often, the common area of these subspaces is chosen as the intersection of the area of best results of the other subspaces.

For example in [JON 00] the subspace varying the number of servers is obtained with segmented access patterns, and with well-chosen block sizes and client:server ratios. Defining such optimal subspaces can be highly system-dependent and may therefore not be as appropriate for b_eff_io designed for a variety of systems. For the design of b_eff_io, it is important to choose the grid points based more on general application needs than on optimal system behavior.

4.2.2 Criteria

The benchmark b_eff_io should characterize the I/O capabilities of the system. Should we use, therefore, only access patterns that promise a maximum bandwidth? No, but there should be a good chance that an optimized implementation of MPI-I/O should be able to achieve a high bandwidth. This means that we should measure patterns that can be recommended to application developers.

An important criterion is that the b_eff_io benchmark should only need about 10 to 15 minutes. For first measurements, it need not run on an empty system as long as concurrently running other applications do not use a significant part of the I/O bandwidth of the system. Normally, the full I/O bandwidth can be reached by using less than the total number of available processors or SMP nodes. In contrast, the communication benchmark b_eff should not require more than 2 minutes, but it must run on the whole system to compute the aggregate communication bandwidth.

Based on the rule for well-balanced systems mentioned in Section 4.2 and assuming that MPI-I/O will attain at least 50 percent of the hardware I/O bandwidth, we expect that a 10 minute b_eff_io run can write or read about 16% of the total memory of the benchmarked system. For this estimate, we divide the total benchmark time into three intervals based on the following access methods: initial write, rewrite, and read.

However, a first test on a T3E900-512 shows that based on the pattern-mix, only about the third of this theoretical value is transferred. Finally, as a third important criterion, we want to be able to compare different common access patterns.

4.3 Definition of the effective I/O bandwidth

The parallel effective I/O bandwidth benchmark measures the following aspects:

- a *set of partitions*;
- the access methods *initial write, rewrite*, and *read*;
- the *pattern types* (see Figure 4.1):
  - (0) strided collective access, scattering large chunks in memory to/from disk,
  - (1) strided collective access, but one read or write call per disk chunk,
  - (2) noncollective access to one file per MPI process, i.e., to separate files,
  - (3) same as (2), but the individual files are assembled to one segmented file,
  - (4) same as (3), but the access to the segmented file is done collectively; files are not reused across pattern types;

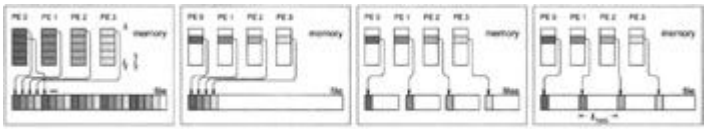


Figure 4.1: Access patterns used in b_eff_io. Each diagram shows the data accessed by **one** MPI-I/O write call

- the contiguous chunk size is chosen *wellformed*, i.e., as a power of 2, and *non-wellformed* by adding 8 bytes to the wellformed size;
- different chunk sizes, mainly 1 kB, 32 kB, 1 MB, and the maximum of 2 MB and 1/128 of the memory size of a node executing one MPI process.

The entire list of access patterns is shown in Table 4.1. The column "type" refers to the pattern type. The column "*l*" defines the size of the contiguous chunks that are written from memory to disk and vice versa. The value M_{PART} is defined as max(*2 MB, memory of one node /128*). This definition should reflect applications that write a significant part of the total memory to disk, but expecting that this is done by several I/O accesses, e.g., by writing several matrices to disk. The smaller chunk sizes should reflect the writing of fixed-sized short (1 kB), middle (32 kB) and long (1 MB) wellformed and non-wellformed (8 bytes longer) data-sets. The column "*L*" defines the size of the contiguous regions accessed at once in memory. In case of pattern type (0), non-contiguous file views are used. If *l* is less than *L*, then in each MPI-I/O read/write call the *L* bytes accessed in memory are scattered/gathered to/from chunks of *l* bytes at various locations on disk. In all other cases, a contiguous chunk written/read on disk has the same size as a contiguous region accessed in memory. This is denoted by "*:=l*" in the *L* column. *U* is a time unit.

Table 4.1: Details of access patterns used in b_eff_io

type	*l*	*L*	*U*
0	1 MB	1 MB	0
	MPART	*:=l*	4
	1 MB	2 MB	4
	1MB	1MB	4
	32 kB	1 MB	2
	1 kB	1 MB	2
	32 kB +8B	1 MB + 256B	2
	1 kB +8B	1 MB + 8 kB	2
	1 MB +8B	1 MB + 8B	2
1	1 MB	*:=l*	0
	MPART	*:=l*	4

	1 MB	:=/	2
	32 kB	:=/	1
	1 kB	:=/	1
	32 kB +8B	:=/	1
	1 kB +8B	:=/	1
	1 MB +8B	:=/	2
2	1 MB	:=/	0
	MPART	:=/	2
	1 MB	:=/	2
	32 kB	:=/	1
	1 kB	:=/	1
	32 kB +8B	:=/	1
	1 kB +8B	:=/	1
	1 MB +8B	:=/	2
3/4	see type=2		
			$\sum U = 64$

Each access pattern is benchmarked by repeating the pattern for a given amount of time. This time is given by the allowed time for a whole partition (e.g., T = 10 minutes) multiplied by $U/\sum U/3$, as given in the table. A value of $U = 0$ indicates that this pattern is executed only once to hide from the benchmark initialization effects of the subsequent patterns. The time-driven approach allows one to limit the total execution time. The total amount of data written to disk is not limited by the chunk sizes. It is only limited by T, the time allowed (= scheduled) for this benchmark, and by the disk-bandwidth of the system that is measured. For pattern types (3) and (4) a fixed segment size must be computed before starting the pattern of these types. Therefore, the time-driven approach is substituted by a size-driven approach, and the repeating factors are initialized based on the measurements for types (0) to (2).

The b_eff_io value **of one pattern type** is defined as the total number of transferred bytes divided by the total amount of time from opening till closing the file. The b_eff_io value **of one access method** is defined as the average of all pattern types with double weighting of pattern type 0.[1] The b_eff_io value **of one partition** is defined as the average of the access methods with the weights 25% for *initial write*, 25% for *rewrite*, and 50 % for *read*. The b_eff_io **of a system** is defined as the maximum over any b_eff_io of a single partition of the system, measured with a scheduled execution time T of at least 15 minutes. This definition permits the user of the benchmark to freely choose the usage aspects and enlarge the total filesize as desired. The minimum filesize is given by the bandwidth for an initial write multiplied by 300 sec (= 15 minutes / 3 access methods). The benchmark can be used on any platform that supports parallel MPI-I/O. On clusters of SMP nodes, the user of the benchmark can decide how many MPI processes should run on each node. The final b_eff_io value is defined as the maximum over all such usage aspects. These aspects must be reported together with the b_eff_io result. For using this benchmark to compare systems as in the TOP 500 list [TOP500] and in the TOP 500 cluster list [TFCC], more restrictive rules are necessary. They are described in Section 4.6.

[1] The double weighting of pattern type 0 and the double weighting of the large chunk sizes (see U = 2 and 4 in Table 4.1) is used to reflect that this benchmark should measure the I/O performance of large systems in high performance computing which typically should be used with large I/O chunks or collective I/O (=pattern type 0).

4.4 Comparing systems using b_eff_io

First, we test b_eff_io on two systems, the Cray T3E900-512 at HLRS/RUS in Stuttgart and an RS 6000/SP system at LLNL called "Blue Pacific". The T3E is an MPP system, the IBM SP is a cluster of SMP nodes. In Section 4.4.1, we will also compare other systems from NEC, Hitachi and IBM, and in Section 4.7, we will show results on an IBM ASCI White system.

On the T3E, we use the tmp-filesystem with 10 striped Raid-disks connected via a GigaRing for the benchmark. The peak-performance of the aggregated parallel bandwidth of this hardware configuration is about 300 MB/s. The LLNL results presented here are for an SP system with 336 SMP nodes each with four 332 MHz processors. Since the I/O performance on this system does not increase significantly with the number of processors on a given node performing I/O, all test results assume that a single thread on a given node is doing the I/O. Thus, a 64 processor run means 64 nodes assigned to I/O, and no requested computation by the additional 64*3 processors.

On the SP system, the data is written to the IBM General Parallel File System (GPFS) called blue.llnl.gov./g/gl which has 20 VSD I/O servers. Recent results for this system show a maximum read performance of approximately 950MB/sec for a 128 node job, and a maximum write performance of 690MB/sec for 64 nodes [JON 00].[2] Note that these are the maximum values observed, and that performance degrades when the access pattern and/or the node number is changed.

For this data on both platforms pre-releases (Rel. 0.x) of b_eff_io were used that had a different weighting of the patterns (type 0, type 1, etc). Therefore the values presented in this section cannot be directly compared with the results in the next section. MPI-I/O was implemented with ROMIO [THA 99, THA1] but with different device drivers.

On the T3E, we have modified the MPI Release mpt. 1.3.0.2, by substituting the ROMIO/ADIO Unix filesystem driver routines for opening, writing, and reading files. The Posix routines were substituted by the asynchronous counter part, directly followed by the wait routine. This combination of asynchronous write (or read) and wait is not semantically identical to the Posix (synchronous) write or read. The Posix semantics does not allow on the T3E that the I/O generated by several processes can be done in parallel on several striping units, i.e., the I/O accesses are serialized.

With the asynchronous counterpart, a relaxed semantics is used which allows parallel disk access from several processes [RAB3]. On the RS 6000/SP Blue Pacific machine, GPFS [GPF 00] is used underneath the MPICH version of MPI with ROMIO. Figure 4.2 shows the b_eff_io values for different partition sizes and different values of *T*, the time scheduled for benchmarking one partition. All measurements were taken in a non-dedicated mode.

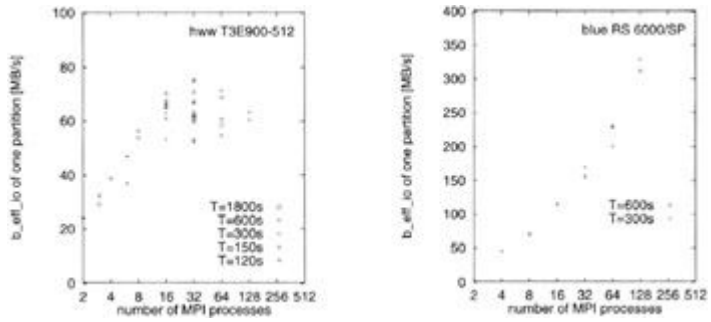


Figure 4.2: Comparison of b_eff_io for different numbers of processes on T3E and SP, measured partially without pattern type 3

Besides the different absolute values that correlate to the amount of memory in each system, one can see very different behavior. For the T3E, the maximum is reached at 32 application processes, with little variation from 8 to 128 processes, i.e., the I/O bandwidth is a global resource. In contrast, on the IBM SP the I/O bandwidth tracks the number of compute nodes until it saturates. In general, an application only makes I/O requests for a small fraction of the compute time. On large systems, such as those at the High-Performance Computing Center in Stuttgart and the Computing Center at Lawrence Livermore National Laboratory, several applications are sharing the I/O nodes, especially during prime time usage. In this situation, I/O capabilities would not be requested by a significant proportion of the CPU's at the same time. "Hero" runs, where one application ties up the entire machine for a single calculation are rarer and generally run during non-prime time. Such hero runs can require the full I/O performance by all processors at the same time. The right-most diagram shows that the RS 6000/SP fits more to this latter usage model. Note that GPFS on the IBM SP is configurable, i.e., number of I/O servers and other tunables, and the performance on any given SP/GPFS system depends on the configuration of that system.

4.4.1 Detailed insight

In this section, we present a detailed analysis of each run of b_eff_io on a partition. For each run of b_eff_io, the I/O bandwidth for each chunk size and pattern is reported in a table that can be plotted as in the graphs shown in each row of Figure 4.3. The three diagrams in each row show the bandwidth achieved for the three different access methods: writing the file the first time, rewriting the same file, and reading it. On each graph, the bandwidth is plotted on a logarithmic scale, separately for

each pattern type and as a function of the chunk size. The chunk size on disk is shown on a pseudologarithmic scale. The points labeled "+8" are the non-wellformed counterparts of the power of two values. The maximum chunk size varies across systems because it is chosen to be proportional to the usable memory size per node to reflect the scaling up of applications on larger systems. On the NEC SX-5, a reduced maximum chunk size was used. Except on the NEC SX-5, we have used b_eff_io releases 1.x. On the IBM SP, a new MPI-I/O prototype was used [PRO 00]. This prototype is used for the development and improvement of MPI-I/O. Performance of the actual product may vary.

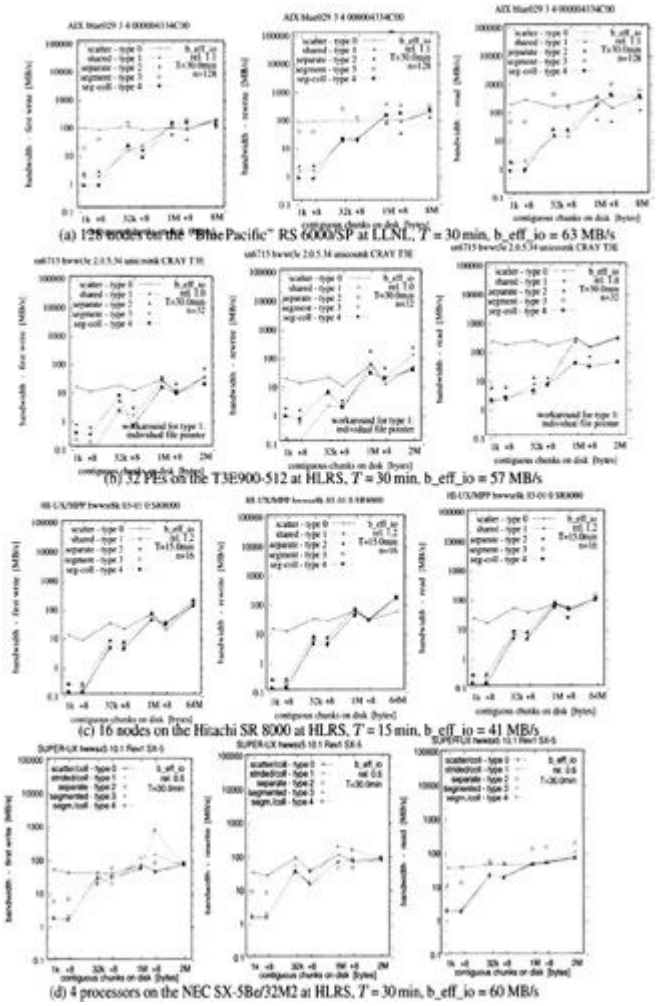


Figure 4.3: Comparison of the results on SP, T3E, SR8000, and SX-5

The four rows compare the I/O bandwidth on four different systems from IBM, Cray, Hitachi and NEC. The IBM SP and the Cray T3E are described in the previous section. The NEC SX-5 system at HLRS is a shared memory vector system. It is a cluster of two 16 processor SMP nodes with 32+48 GB memory, but the benchmark was done only on a single SMP node. It has four striped RAID-3 arrays DS 1200, connected by fibre channel. The NEC Supercomputer File System (SFS) is used. It is optimized for large-sized disk accesses. The filesystem parameters are: 4 MB cluster size (=block size), and if the size of an I/O request is less than 1 MB, then a 2 GB filesystem-cache is used. On the NEC SX-5, we use MPI/SX 10.1. The Hitachi SR8000 at HLRS is a 16 node system, that clusters SMP nodes each with 8 pseudo-vector CPUs and with 8 GB of memory.

First, notice that access pattern type 0 is the best on all platforms for small chunk sizes on disk. Thus all MPI-I/O implementations can effectively handle the 1 MB memory regions that are given in each MPI-I/O call to be scattered to disk or gathered from disk. In all other pattern types, the memory region size per call is identical to the disk chunk size, i.e., in the case of 1 kB or 32 kB, only a small or medium amount of data is accessed per call.

Note that due to the logarithmic scale, a vertical difference of only a few millimeters reflects an order of magnitude change. Comparing the wellformed and non-wellformed measurements, especially on the T3E, there are huge differences. Also on the T3E, we see a large gap between write and read performance in the scattering pattern type.

On the IBM SP MPI-I/O prototype, one can see that segmented non-collective pattern type 3 is also optimized. On the other hand, the collective counterpart is more than a factor of 10 worse. Such benchmarking can help to uncover advantages and weakness of an I/O implementation and can therefore help in the optimization process. Figure 4.4 and the first row of Figure 4.3 compare two different MPI-I/O implementations on top of the GPFS filesystem. ROMIO is used in the benchmarks shown in Figure 4.4. One can see that with ROMIO writing and reading of separate files result in best bandwidth values. IBM MPI-I/O prototype clearly shows in Figure 4.3 better results for access pattern type 0 and for access pattern type 3. File hints, introduced by the MPI-2 standard with the *info* argument, can be used in the IBM MPI-I/O prototype [PRO 00] in order to further optimize I/O performance for speficic access patterns. These hints are necessarily pattern specific, since if they worked for all pattern types, they would naturally be a part of the standard MPI-I/O implementation. In Section 4.7, we present how the next release of b_eff_io will enable the selective use of MPI-I/O implementation specific file hints on a per access pattern basis, and we illustrate the impact of using such hints by providing early results using the latest version of the IBM

MPI-I/O prototype [PRO 01] on an ASCI White system.

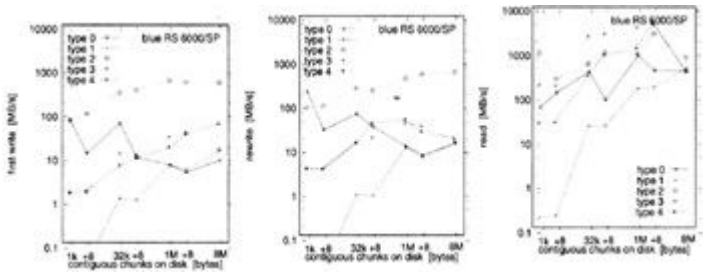


Figure 4.4: 128 nodes on the Blue Pacific RS 6000/SP with ROMIO

Figure 4.5 compares the final b_eff_io results on these four platforms. Values for other partition sizes are added. It should be noticed that the IBM results in this figure can not be compared with the results in Figure 4.2, because for Figure 4.5, ROMIO and b_eff_io rel. 0.x are used instead of the IBM MPI-I/O prototype and b_eff_io rel. 1.1. In particular, b_eff_io rel. 0.x used an inappropriate weighting to compute the bandwidth average. Consequently, the absolute values were not comparable between different platforms. This problem was fixed in rel. 1.0. With the IBM MPI-I/O prototype, the bad effective I/O bandwidth for small partition sizes (as shown with ROMIO in Figure 4.2) could also be fixed.

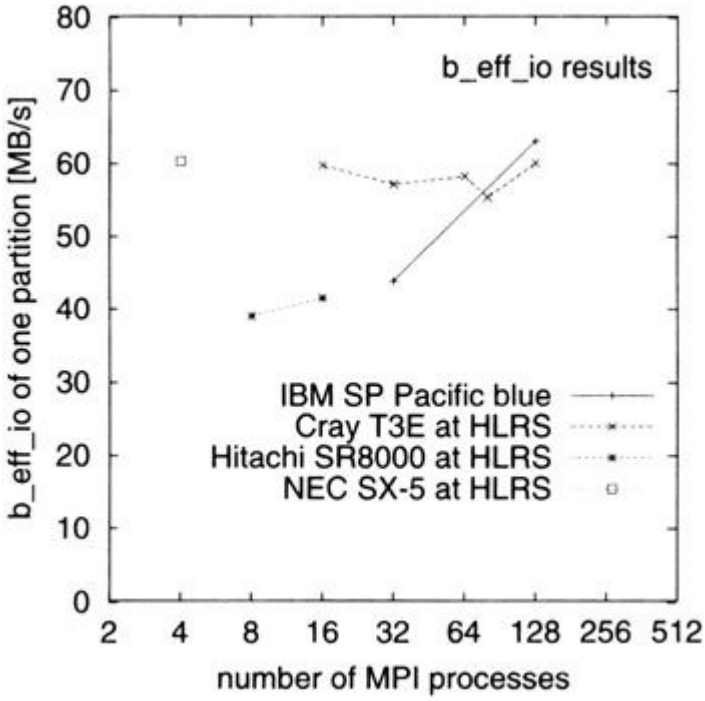


Figure 4.5: Comparison of b_eff_io for various numbers of processes at HLRS and LLNL, measured partially without pattern type 3. b_eff_io releases 1.x were used, except for the NEC system (rel. 0.6)

In general, our results show that the b_eff_io benchmark is a very fast method to analyze the parallel I/O capabilities available for applications using the standardized MPI-I/O programming interface. The resulting b_eff_io value summarizes parallel I/O capabilities of a system in one significant I/O bandwidth value.

[2]Upgrades to the AIX operating system and underlying GPFS software may have altered these performance numbers slightly between measurements in [JON 00] and in the current work. Additionally, continual upgrades to AIX and GPFS are bringing about improved performance overall.

4.5 Discussion of b_eff_io

In this section, given the primary results of the benchmark, we reflect on some details of its definition. The design of the b_eff_io tries to follow the rules about MPI benchmarking defined by Gropp and Lusk [GRO 99], as well as Hempel [HEM 99], but there are a few problematic issues.

Normally, the same experiment should be **repeated** a few times to compute a **maximum bandwidth**. To achieve a very fast I/O benchmark suite, this methodology is substituted by **weighted averaging** over a medium number of experiments, i.e., the access patterns. (We note that in the case of the IBM SP data, repeated calculation of b_eff_io on different days produced nearly identical answers). The weighted averaging is done for each experiment after calculating the average bandwidth over all repetitions of the same pattern. Any maximum is calculated only after repeating the total b_eff_io benchmark itself. For this maximum, one may vary the number of MPI processes (i.e., the partition size), the schedule time T, and filesystem parameters.

The major problem with this definition is that one may use any schedule time T with $T > 10$ minutes. First experiments on the T3E have shown that the b_eff_io value may have its maximum for $T = 10$ minutes. This is likely because for any larger time interval the caching of the filesystem in the memory is reduced.

Indeed, **caching issues** may be problematic for I/O benchmarks in general. For example, Hempel [HEM 00] has reported that on NEC SX-5 systems other benchmark programs have reported a bandwidth significantly higher than the hardware peak performance of the disks. This is caused by a huge 4 GB memory cache used by the filesytem. In other words, the measurement is not able to guarantee that the data was actually written to disk. To help assure that data is written, we can add MPI_File_sync. The problem is, however, that MPI_File_sync influences only the consistency semantics. Calling MPI_File_sync after writing to a file guarantees that any other process can read this newly written data, but it does **not** guarantee that the data is stored on a permanent storage medium, i.e., that the data is written to disk. There is only one way to guarantee that the MPI-I/O routines have stored 95 % of the written data to disk. One must write a dataset 20 times larger than the memory cache length of the filesystem. This can be controlled by verifying that the datasize accessed by each b_eff_io access method is larger than 20 times of the filesystems' cache length.

The next problem arises from the **time driven approach** of the b_eff_io benchmark. Each access pattern is repeated for a given time interval, which is $T_{pattern} = T/3 * U/\sum U$ for each pattern. The termination condition must be computed after each call to a write or read routine. In all patterns defining a collective fileview or using collective write or read routines, the termination condition must be computed globally to guarantee that all processes are stopped after the same iteration. In the current version, this is done by computing the criterion only at a root process. The local clock is read after a barrier synchronization. Then, the decision is broadcasted to all other nodes. This termination algorithm is based on the assumption that a barrier followed by a broadcast is at least 10 times faster than a single read or write access. For example, the fastest access on the T3E for $L = 1$ kB regions is about 4 MB/s, i.e., 250 μs per call. In contrast, a barrier followed by a broadcast needs only about 60 μs on 32 PEs, which is not 10 times faster than a single I/O call. Therefore, this termination algorithm should be modified in future versions of our benchmark. Instead of computing the termination criterion at the end of each iteration, a geometric series of increasing repeating factors should be used: The repeating factor used in the first iteration must be small, because otherwise the execution time of the first iteration may be larger than the allowed time (= $U/\sum U/3$, see Section 4.3) on some slow platforms. After benchmarking this first iteration, the repeating factor can be increased to reduce the relative benchmarking overhead induced by the time measurements and barrier operation at the end of each iteration.

Pattern types 3 and 4 require a predefined **segment size** L_{SEG} (see Figure 4.1). In the current version, for each chunk size "l", a repeating factor is calculated from the measured repeating factors of pattern types 0-2. The segment size is calculated as the sum of the chunk sizes multiplied by these repeating factors. The sum is rounded up to the next multiple of 1 MB. This algorithm has two drawbacks:

1. The alignment of the segments are multiples of 1 MB. If the striping unit is more than 1 MB, then the alignment of the segments is not wellformed.

2. On systems with **32 bit integer/int** datatype, the segment size multiplied by the number of processes (n) may be more than 2 GB, which may cause internal errors inside of the MPI library. Without such internal restrictions, the maximum segment size would be $16/n$ GB, based on a 8 byte element type. If the segment size must be reduced due to these restrictions, then the total amount of data written by each process does not fit any longer into one segment.

On large MPP systems, it may also be necessary to reduce the **maximum chunk size** (M_{PART}) to 2/n GB or 16/n GB. This restriction is necessary for pattern types 0, 1, 3 and 4.

Another aspect is the mode used to open the benchmark files. Although we want to benchmark **unique** mode, i.e., ensure that a file is not accessed by other applications while it is open by the benchmark program, MPI_MODE_UNIQUE_OPEN must **not** be used because it would allow an MPI-I/O implementation to delay all MPI_File_sync operations until the closing of the file.

4.6 The time-driven approach

Figure 4.2 shows interesting results. There is a difference between the maximum I/O bandwidth and the sampled bandwidth for several partition sizes. In the redesign from release 0.x to 1.x, we have incorporated that the averaging for each pattern type can not be done by using the average of the bandwidth values for all chunk sizes. The bandwidth of one pattern type must be computed as the total amount of transferred data divided by the total amount of time used for all chunk sizes. With this approach, it is possible to reduce caching effects and to allow a total scheduled time of 30 minutes for measuring all five patterns with the three access directions (write, rewrite, read) for **one** compute partition size.

The b_eff_io benchmark is proposed for the *Top 500 Clusters* list [TFCC]. For this, the I/O benchmark must be done automatically in 30 minutes for **three** different compute partition sizes. This can be implemented by reorganizing the sequence of the experiments. First, all files are written with the three different compute partition sizes, followed by rewriting, and then by all reading. Additionally, the rewriting experiments only use pattern type 0, to reduce the amount of time needed for each partition size without losing the chance to compare the *initial write* with the *rewrite* pattern. Therefore, the averaging process has also to be slightly modified. The average for writing patterns is done by weighting all 5 *initial write* patterns and the one *rewrite* pattern each with a factor of one. This implies that pattern type 0 is weighted double, as it is also done with the *read* pattern types. The b_eff_io value is then defined as the geometric mean of the writing and reading average bandwidth values. The geometric mean is used to guarantee that both bandwidth values (writing and reading) influence the final result in an appropriate way, even if one of the two values is very small compared to the other one. The arithmetic mean was not chosen, because it would always report a value larger than 50 % of the higher bandwidth value, even if the other bandwidth is extremely low.

Remembering that the b_eff_io benchmark has two goals, (a) to achieve a detailed insight into the performance of several I/O patterns, and (b) to summarize these benchmarks in one specific effective I/O bandwidth value, we offer the option to run this b_eff_io release 2.0 benchmark for a longer time period and with all rewriting patterns included. In this case, the scheduled time is enlarged to 45 minutes. The averaging process is not changed, i.e., the additional *rewriting* patterns do not count for the b_eff_io number, but with this option the benchmark can still be used to compare all *initial write* patterns with their *rewrite* counterparts.

For using b_eff_io as an additional metric in the Top 500 Clusters list, it is also necessary to define three partition sizes to get comparable results and to reduce the total time to run this benchmark on a given platform. The three partition sizes, i.e., the number of nodes of a cluster or the number of CPUs of an MPP or SMP system, are given by the following formulas:

- small = 2^{**} (round (log_2(SIZE) * 0.35))
- medium = 2^{**} (round (log_2(SIZE) * 0.70))
- large = SIZE,

with SIZE = size of MPI_COMM_WORLD and the following exceptions: small=1 if SIZE==3., and medium=3 if SIZE==4.[3] The three partition sizes should allow to analyze the behavior of the system when applications try to make I/O by using different numbers of CPUs as shown in Figure 4.2 and in Section 4.4.

The size of MPI_COMM_WORLD should reflect the total cluster system. If the system is used on a cluster of SMP nodes, it must be reported how many CPUs per node were used to run this benchmark.

[3]The three values (35 %, 70 %, and 100 %) are motivated as a (nearly) linear splitting of the total number in the logarithmic scale. The zero value (i.e. 2^0 = 1 CPU) is omitted, because large parallel systems are normally not dedicated for serial applications.

4.7 The influence of file hints

The MPI-I/O interface allows for the specification of file hints to the implementation so that an optimal mapping of the application's parallel I/O requests to the underlying filesystem and to the storage devices below can be carried out. First benchmark tests with b_eff_io have shown a significant difference in the behavior of two different MPI-I/O implementations on top of the same filesystem. IBM's prototype implementation shows in the first row of Figure 4.3 an optimum with the access pattern type 0, while ROMIO yields the best results with the access pattern type 2 (Figure 4.4). IBM Research [PRO 00, PRO 01] developed an MPI-I/O prototype, referred to as MPI-IO/GPFS and using GPFS as the underlying file system, in order to investigate how file hints can be used to optimize the performance of various parallel I/O access patterns. They demonstrate important benefit of using file hints for specific access patterns and access methods (reading vs. writing).

We present here how the b_eff_io benchmark can be enhanced in order to enable the use of implementation specific MPI-I/O hints on a per access pattern type basis. We first describe our methodology. Then, we detail our on-going experimentation aimed at applying this methodology on the ASCI White RS 6000/SP system at Lawrence Livermore National Laboratory, using the latest version of IBM's MPI-IO/GPFS prototype. Preliminary results from this experimentation are then presented and demonstrate how the use of hints can lead to improved parallel I/O performance (i.e., higher b_eff_io numbers).

4.7.1 Support for file hints

The support for file hints can be done with an additional options file, which specifies file hints to be set at file open time for each pattern type and each access method. Additionally, for each disk chunk size used, separate hints can be given. For some pattern types, these size dependent hints may be set together with the definition of an additional fileview.

The usage of such file hints is automatically reported together with the benchmark results, allowing the user to get additional information about the optimal usage of the MPI-I/O implementation for specific access patterns, access methods, and disk chunk sizes.

4.7.2 Experimentation with IBM's MPI-IO/GPFS prototype

The foundation of the design of MPI-IO/GPFS is the technique referred to as data shipping. This technique prevents GPFS file blocks to be accessed concurrently by multiple tasks, possibly residing on separate nodes. Each GPFS file block is bound to a single I/O agent, which is responsible for all accesses to this block. For write operations, each task distributes the data to be written to the I/O agents according to the binding scheme of GPFS file blocks to I/O agents. I/O agents in turn issue the write calls to GPFS. For reads, the I/O agents read the file first, and ship the data read to the appropriate tasks. The binding scheme implemented by MPI-IO/GPFS consists in assigning the GPFS file blocks to a set of I/O agents according to a round-robin striping scheme, illustrated in Figure 4.6. I/O agents are also responsible for combining data access requests issued by all participating tasks in collective data access operations.

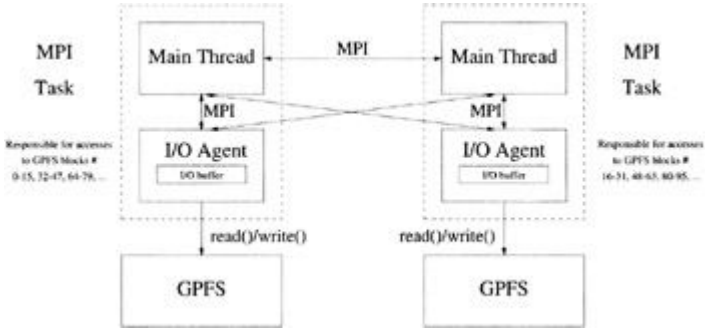


Figure 4.6: GPFS block allocation used by MPI-IO/GPFS in data shipping mode (using the default stripe size)

On a per open file basis, the user can define the stripe size used in the binding scheme of GPFS blocks to I/O agents, by setting file hint *IBM_io_buffer_size*. The stripe size also controls the amount of buffer space used by each I/O agent in data acccess operations. Its default size is the number of bytes contained in 16 GPFS file blocks.

The user can enable or disable the MPI-IO data shipping feature, by setting file hint *IBM_largeblock_io* to "*false*" or "*true*", respectively. Data shipping is enabled by default. When it is disabled, tasks issue read/write calls to GPFS directly. This saves the cost of transferring data between MPI tasks and I/O agents, but risks GPFS file block ping-ponging across nodes if tasks located on distinct nodes contend for GPFS file blocks in read-write or write-write shared mode (since GPFS performs coherent client caching across nodes). In addition, collective data access operations are implemented as noncollective operations when data shipping is disabled.

Therefore, it is recommended to disable data shipping on a file only when accesses to the file are performed in large chunks

of when tasks access large disjoint partitions of the file. In such cases, MPI-IO coalescing of the I/O requests of a collective data access operation cannot provide benefit and GPFS file block access contention is not a concern.

For these reasons, it seems natural to set the *IBM_largeblock_io* hint to true if the access pattern is segmented (like for access pattern types 2, 3, and 4). For scattering access pattern types 0 and 1, it also seems natural to set *IBM_largeblock_io* to true for larger values of *l* (e.g., 1MB and M_{PART}), and leave it to false for smaller values of *l* (e.g., less than 1MB). Finally, in the latter case (pattern types 0 and 1, and smaller values of *l*), it is certainly interesting to experiment with various values for the *IBM_io_buffer_size* hint, one below the default value (e.g., a size of 4 GPFS file blocks), and one above the default value (e.g., a size of 64 GPFS file blocks). To use different file hints for the different patterns and chunk sizes, the hint value must be set to "*switchable*" on file open. The hints described must be specified when the file view is set for each pattern.

MPI-I/O-GPFS allows one also to define the hint *IBM_sparse_access*, which can be set to true if the access pattern is collectively sparse (this does not apply to any pattern type used by b_eff_io); by default, the value of this hint is false. This hint is not used in the b_eff_io benchmark.

Our current experimentation is performed using IBM MPI-IO/GPFS on an ASCI White RS 6000/SP testbed system at Lawrence Livermore National Laboratory. This IBM SP system is composed of 67 SMP nodes each consisting of 16 375 MHz Power3 processors. There are 64 compute nodes, 1 login node, and 2 dedicated GPFS server nodes. Each node has 8 GB of memory. The GPFS configuration uses a GPFS file block size of 512 KB and has a pool of 100 MB on each of the nodes. It uses two VSD servers. Each VSD server serves 12 logical disks (RAID5 sets). The filesystem is comprised of 24 logical disks, of about 860 GB each. The transfer rate of the disks integrated as one system is approximately 700 MB/sec.

For these tests, a partition of 16 nodes is used with 4 MPI processes per node. b_eff_io 2.0 prototype is used. Two sets of file hints are used. The first set uses all default (IBM_largeblock_io = false and IBM_io_buffer_size = 8MB) file hint values. This is our **baseline**.

The **second set** has the IBM_largeblock_io hint set to true. For pattern type 2 (*separated files*), the mode MPI_MODE_UNIQUE_OPEN is removed for both measurements.

The **third set** has again the IBM_largeblock_io hint set to true and IBM_io_buffer_size hint set to the default (8 MB) for all patterns, except on the pattern types *scatter* and *shared* with the chunk sizes 1 kB, 1 kB + 8 bytes, 32 kB, and 32 kB + 8 bytes, where IBM_largeblock_io = false and IBM_io_buffer_size = 2 MB is used[4]. T is chosen to be 30 minutes per set of file hints.

4.7.3 Results with hints enabled

The results are plotted in Figure 4.7, Table 4.2, and Table 4.3. In Figure 4.7, we can compare the baseline benchmarks (without hints) with the results for the second and third set of hints. The first row (a) compares the benchmark results for the pattern types 0 (*scatter*). We can see that the second set[5] improves the results for the large chunk sizes whereas we can see that for the chunk sizes less than 1 MB, the baseline results are clearly better. The right diagram uses the third set of hints. For the chunk sizes less than 1 MB, the hints are switched to IBM_largeblock_io = false and IBM_io_buffer_size=2MB. Comparing this set of hints with the baseline, we can see that the bandwidth could be improved for all chunk sizes, except at writing of non-wellformed chunks with 1 MB plus 8 bytes. The second row (b) shows for pattern types 1 (*shared*) that the hints give only a real advantage for 1 MB wellformed chunks. Rows (c) and (d) show the other pattern types, row (c) with the baseline and row (d) with the third set of hints.[6]

Table 4.2: Summary of the effect of hints on the total bandwidth on the ASCI White testbed

Access	set of hints	IBM_large-block_io	scatter type 0 MB/s	shared type 1 MB/s	separate type 2 MB/s	segmented type 3 MB/s	seg-coll type 4 MB/s	weighted average MB/s
write	baseline	false	197	188	104	338	13	173
	second	true	284	277	435	541	524	391
	third	mixed	397	282	423	533	502	422
rewrite	baseline	false	190	172	83	186	11	139
	second	true	197	244	241	228	431	256
	third	mixed	449	265	234	297	297	332
read	baseline	false	122	130	96	408	11	148
	second	true	483	346	609	586	590	516
	third	mixed	441	282	579	554	544	473
average	baseline	false	157	155	95	335	12	152

	second	true	362	303	473	485	534	420
	third	mixed	432	278	454	485	472	425

Table 4.3: b_eff_io without and with IBM_largeblock_io hint on the ASCI White testbed

set of hints	IBM_largeblock_io	b_eff_io MB/s
baseline	false (default)	159.5
second	true	440.8
third	mixed	451.7

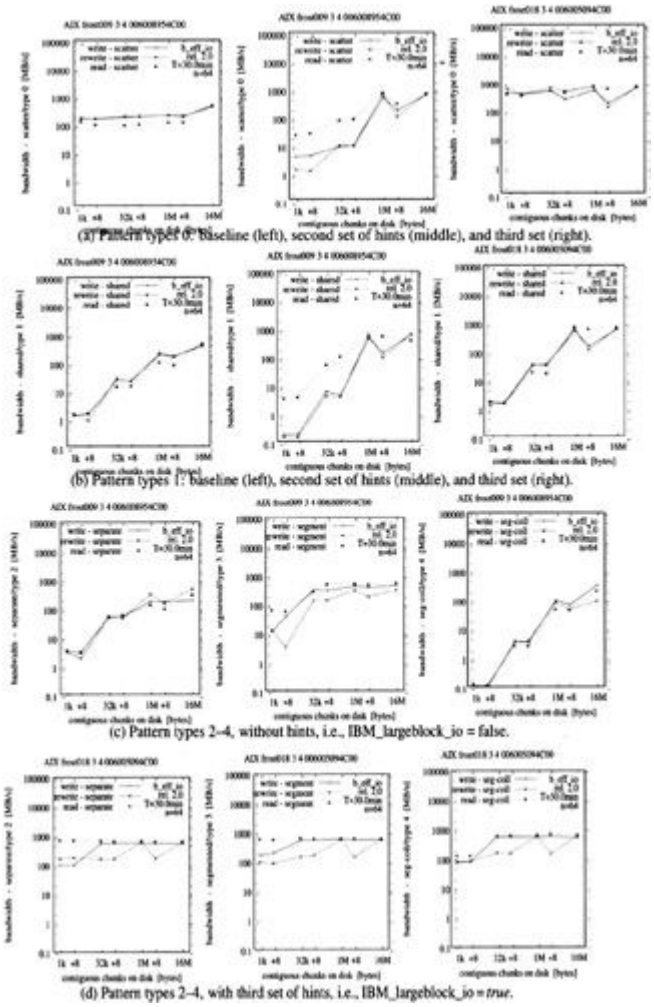


Figure 4.7: The effect of hints on the ASCI White RS 6000/SP testbed at LLNL

The left diagram in rows (c) and (d) show pattern types 2 (*separated*). The hint improves the bandwidth in all disciplines. Looking at the *segmented* pattern type (3) in the middle diagrams one can see that the baseline could be improved by the hint mainly for short chunk sizes and read operation. In the collective case (type 4=*segcoll*) in the right hand diagrams the baseline produces very poor results while the benchmark with hints enabled runs with good bandwidth. The curves clearly show that the third settings for the hints[7] have produced the best results for each chunk size.

The result for each pattern type and each access is summarized in a total bandwidth shown in Table 4.2. The total bandwidth is defined as the total number of transferred bytes divided by the total amount of time from opening till closing the file. The table shows, that in all cases, the hint IBM_largeblock_io = true improves the baseline result (IBM_largeblock_io=false). This means that some losses with IBM_largeblock_io = true in the area of chunk sizes less than 1 MB are more than only compensated by the advantages of this hint for larger chunk sizes. The weighted average in the last column is presented as an arithmetic mean with the double weighted scatter pattern according to the time units used for the benchmarking. The average at the bottom row is presented according to the formula ((write+rewrite)/2+read)/2 to guarantee that the write and rewrite patterns have not more weight than the read patterns. Looking at the 15 values representing the 5 different pattern types and the 3 access methods, the table shows that with the hint set to true, all bandwidth values are equal or larger than 197 MB/s. In the baseline results, only three values have reached this value of 197 MB/s. With this hint, the average line (bottom row) shows bandwidth values larger than 300 MB/s for all pattern types.

With the third set of hints, again the result could be improved mainly for type 0. For type 1, we can see a small reduction. The results for the second and third sets also show that the detailed values are not exactly reproducible and that there may be a minor reduction of bandwidth due to the overhead for allowing hints to be switchable.

Table 4.3 shows that the b_eff_io value could be improved from 159.5 MB/s for the baseline to 440.8 MB/s if the IBM_largeblock_io hint is enabled and to 451.7 MB/s with the third set of hints. Note, that in b_eff_io version 2.0, the definition of the final b_eff_io had to be slightly modified to allow fast benchmarking without measuring all rewrite patterns. In the new formula, b_eff_io is the geometric mean of the writing and reading average. The reading average was not changed. The writing average is the sum of the bandwidth values of *all write* pattern types plus the *rewrite scatter* pattern type, divided by 6. This means that in the reading and in the writing average, the *scatter* pattern type is double weighted, in the first case by using the *write* and the *rewrite* results, in the second case by counting the *read-scatter* result twice in the average.[8]

This section has shown that hints can significantly improve the parallel I/O bandwidth. This benchmark can be used to test the hints and can assist the process of defining an optimal default set of hints.

[4]For this set of hints, the hints must be set to *switchable* when opening the file - it takes the default value on opening the file and any subsequent call to MPI_FILE_SET_VIEW or MPI_FILE_SET_INFO allows to switch it to *true/false* as often as required.

[5]IBM_largeblock_io = true and IBM_io_buffer_size = 8 MB (default) for all patterns.

[6]The second set is omitted, because it uses the same hints as in the third set, except that they are not declared as switchable, and because it exhibits only minor differences from the results of the third set.

[7]IBM_largeblock_io is set to false and IBM_io_buffer_size is reduced to 2 MB for the pattern types 0 (*scatter*) and 1 (*shared*) if the chunk size is less than 1 MB, otherwise IBM_largeblock_io = true and IBM_io_buffer_size = 8 MB.

[8]The value according to b_eff_io version 1.x is computed in the lower right corner of Table 4.2.

4.8 Future work

We plan to use this benchmark to compare several additional systems. In a next stage, cluster type supercomputers should be compared with Beowulf type systems, built with off-the-shelf components and operated with a parallel filesystem, e.g., with the Parallel Virtual File System (PVFS) [CAR 00].

Although [CRA 95] stated that "the majority of the request patterns are sequential", we should examine whether random access patterns can be included into the b_eff_io benchmark. The b_eff_io benchmark was mainly developed to benchmark parallel I/O bandwidth, but it may be desirable to include also a shared data access, i.e., reading the same data from disk into memory locations on several processes; this pattern type is more like checking whether the optimization of the parallel MPI-I/O can detect such a pattern and can implements it by a single reading from disk (e.g., divided into several portions accessed by several processes) and then broadcasting the information to the other processes.[9]

The benchmark will also be enhanced to write an additional output that can be used in the SKaMPI *comparison page* [REU 98]. This combination should allow one to use the same web interface for reporting I/O bandwidth (based on this b_eff_io) together with communication bandwidth (based on SKaMPI benchmark or on the parallel effective communication benchmark b_eff [RAB1, RAB 01]).

[9]This type of a *shared* access pattern should not be confused with the *shared* filepointer used in pattern type 1.

4.9 Conclusion

In this paper, we have described in detail b_eff_io, the parallel effective I/O bandwidth benchmark. We used this benchmark to characterize the I/O performance of common computing platforms. We have shown how this benchmark can provide both detailed insight into the I/O performance of high-performance platforms and how this data can be reduced to a single number averaging important information about that system's accumulated I/O bandwidth. We gave suggestions for interpreting and improving the benchmark, and for testing the benchmark on one's own system. We also showed how the power of MPI-I/O file hints can be taken advantage of on a selective basis in the benchmark. Although we have used the benchmark on small clusters with Linux and other operating systems, we typically find that the parallel filesystems on these clusters are not yet sufficiently advanced to support the full MPI-I/O standard efficiently. We expect that benchmarks such as the one described in this paper will help to spur the design of these systems and that the situation should remedy itself soon. The b_eff_io benchmark can be used together with the Linpack benchmark [TOP500] and the effective communication bandwidth benchmark b_eff [KON 01] to measure the balance of the accumulated computational speed (Linpack R_{max}), the accumulated communication bandwidth (b_eff) and the total I/O bandwidth (b_eff_io).

The authors would like to acknowledge their colleagues and all the people that supported this project with suggestions and helpful discussions. Work at LLNL was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

4.10 Bibliography

[CAR 00] Carns P. H., Ligon W. B. III, Ross R. B., and Thakur R., *"PVFS: A Parallel File System For Linux Clusters"*, *Proceedings of the 4th Annual Linux Showcase and Conference*, Atlanta, GA, October 2000, p. 317-327, http://parlweb.parl.clemson.edu/pvfs/.

[CAR 92] Carter R., Ciotti R., Fineberg S., and Nitzberg W., *"NHT-1 I/O Benchmarks"*, *Technical Report RND-92-016*, NAS Systems Division, NASA Ames, November 1992.

[CRA 95] Crandall P., Aydt R., Chien A., and Reed D., *"Input-Output Characteristics of Scalable Parallel Applications"*, *Proceedings of Supercomputing '95*, ACM Press, Dec. 1995, http://www.supercomp.org/sc95/proceedings/ .

[DET 98] Detert U., *"High-Performance I/O on Cray T3E"*, *40th Cray User Group Conference*, June 1998.

[DIC 98] Dickens P. M., *"A Performance Study of Two-Phase I/O"*, *Proceedings of Euro-Par '98, Parallel Processing*, Southampton, UK, 1998, D. Pritchard, J. Reeve Eds., LNCS 1470, p. 959-965.

[GPF 00] *IBM General Parallel File System for AIX: Installation and Administration Guide*, IBM Document SA22-7278-03, July 2000.

[GRO 99] Gropp W. and Lusk E., *"Reproducible Measurement of MPI Performance Characteristics"*, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proceedings of EuroPVM/MPI '99*, Barcelona, Spain, Sep. 26-29, 1999, J. Dongarra et al. Eds., LNCS 1697, p. 11-18, http://www.mcs.anl.gov/mpi/mpptest/hownot.html.

[HAS 98] Haas P.W., *"Scalability and Performance of Distributed I/O on Massively Parallel Processors"*, *40th Cray User Group Conference*, June 1998.

[HEM 99] Hempel R., *"Basic Message Passing Benchmarks, Methodology and Pitfalls"*, *SPEC Workshop on Benchmarking Parallel and High-Performance Computing Systems*, Wuppertal, Germany, Sep. 13, 1999, http://www.hlrs.de/mpi/b_eff/hempel_wuppertal.ppt.

[HEM 00] Hempel R., and Ritzdorf H., *"MPI/SX for Multi-Node SX-5"*, *SX-5 Programming Workshop*, High-Performance Computing Center, University of Stuttgart, Germany, Feb. 14-17, 2000, http://www.hlrs.de/news/events/2000/sx5.html.

[HO 99] Ho R. S. C., Hwang K., and Jln H., *"Single I/O Space for Scalable Cluster Computing"*, *Proceedings of 1st IEEE International Workshop on Cluster Computing* (*IWCC '99*), Melbourne, Australia, Dec. 2-3, 1999, p. 158-166, http://andy.usc.edu/papers/IWCC'99.pdf.

[JON 00] Jones T., Koniges A. E., and Yates R. K., *"Performance of the IBM General Parallel File System"*, *Proceedings of the International Parallel and Distributed Processing Symposium*, May 2000. Also available as UCRL JC135828.

[KOE 98] Koeninger K., *"Performance Tips for GigaRing Disk I/O"*. *40th Cray User Group Conference*, June 1998.

[KON 01] Koniges A. E., Rabenseifnerr, and Solchenbach K., *"Benchmark Design for Characterization of Balanced High-Performance Architectures"*, *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS '01), Workshop on Massively Parallel Processing (WMPP)*, San Francisco, CA, Apr. 23-27, 2001, IEEE Computer Society Press.

[LAN 98] Lancaster D., Addison C., and Oliver T., *"A Parallel I/O Test Suite"*, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proceedings of EuroPVM/MPI '98*, Liverpool, UK, Sep. 1998, V. Alexandrov, J. Dongarra Eds., LNCS 1497, p. 36-44, 1998, http://users.wmin.ac.uk/~lancasd/MPI-IO/mpi-io.html.

[MPI 97] Message Passing Interface Forum, *MPI-2: Extensions to the Message-Passing Interface*, July 1997, http://www.mpi-forum.org.

[PRO 00] Prost J.-P., Treumann R., Hedges R., Jia B., Koniges A. E., and White A., *"Towards a High-Performance Implementation of MPI-IO on top of GPFS"*, *Proceedings of Euro-Par 2000*, Munich, Germany, Aug. 29 - Sep. 1, 2000.

[PRO 01] Prost J.-P., Treumann R., Hedges R., Jia B., and Koniges A. E., *"MPI-IO/GPFS, an Optimized Implementation of MPI-IO on top of GPFS"*. *Proceedings of Supercomputing '01*, Denver, CO, Nov 11-16, 2001.

[RAB 01] Rabenseifner R., and Koniges A.E., *"Effective Communication and File-I/O Bandwidth Benchmarks"*, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proceedings of EuroPVM/MPI 2001*, Y. Cotronis, J. Dongarra Eds., LNCS 2131, p. 24-35, 2001.

[RAB 1] Rabenseifner R., *Effective Bandwidth (b_eff) Benchmark*, http://www.hlrs.de/mpi/b_eff/ .

[RAB2] Rabenseifner R., *Effective I/O Bandwidth (b_eff_io) Benchmark*, http://www.hlrs.de/mpi/b_eff_io/ .

[RAB3] Rabenseifner R., *Striped MPI-I/O with mpt.1.3.0.1*, http://www.hlrs.de/mpi/mpi_t3e.html#StripedIO.

[REU 98] Reussner R., Sanders P., Prechelt L., and Müller M., *"SKaMPI: A detailed, accurate MPI benchmark"*, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proceedings of EuroPVM/MPI '98*, 1998, LNCS 1497, p. 52-59, 1998, http://www.ipd.ira.uka.de/~skampi/

[SOL 99] Solchenbach K., *"Benchmarking the Balance of Parallel Computers"*, *SPEC Workshop on Benchmarking Parallel and High-Performance Computing Systems*, Wuppertal, Germany, Sep. 13, 1999,

[SOL 1] Solchenbach K., Plum H.-J., and Ritzenhoefer G., *Pallas Effective Bandwidth Benchmark - Source Code and Sample Results*, ftp://ftp.pallas.de/pub/PALLAS/PMB/EFF_BW.tar.gz .

[THA 99] Thakur R., Gropp W., and Lusk E., *"On Implementing MPI-IO Portably and with High Performance"*, *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems (IOPADS '99)*, May 1999, p. 23-32.

[THA1] Thakur R., Lusk E., and Gropp W., *ROMIO: A High-Performance, Portable MPI-IO Implementation*, http://www.mcs.anl.gov/romio/.

[TFCC] TFCC - IEEE Task Force on Cluster Computing, http://www.ieeetfcc.org, and *Top Clusters*, http://www.TopClusters.org.

[TOP500] Universities of Mannheim and Tennessee, *TOP500 Supercomputer Sites*, http://www.top500.org .

Chapter 5: Parallel Join Algorithms on Clusters

Peter Kirkovits, Erich Schikuta, Institut für Informatik und Wirtschaftsinformatik, University of Vienna, Rathausstraße 19/9, A-1010 Vienna, Austria

5.1 Introduction

The join is the most important, but also the most time consuming operation in relational database systems. We implemented four different parallel Join algorithms (Grace Join, Hybrid Join, Nested Loop Join, Sort Merge Join) on a PC-cluster architecture and analyzed their performance behavior. We show that off-the-shelf, cost saving, cluster systems can build a viable platform for parallel database systems.

Today novel applications with dramatically increasing data volumes in the range of Petabytes demand increasing computational power. In these problem domains relational or object-relational database technology is judged as the only suitable platform due to its undeniable expressive power and theoretical soundness. A typical example for this development are high energy physics applications (see [SEG 00]). Therefore much research has focused on the exploitation of parallel and distributed IO techniques in the last few years (for a survey see [ABD 98]).

Due to its immense costs and difficult manageability the development of classical, special-purpose parallel database machines was replaced by the use of affordable and available distributed architectures. This trend led also to a shift of the research in the programming paradigms of database operations. Thus smart but complex parallel approaches are neglected to relatively simple distributed schemes (very common in today's commercial database systems), which leads apparently to loss in possible performance gains.

In the light of this situation we see a dramatic need for research on "classical" parallel operational schemes focusing distributed architectures now. Due to its inherent expressive power the most important operation in a relational database system is the join. It allows one to combine information of different relations according to a user specified condition, which makes it the most demanding operation of the relational algebra. Thus the join is obviously the central point of research for performance engineering in database systems.

Generally, two approaches for parallel or distributed execution of database operations can be distinguished, inter- and intra-operational parallelism. Inter-operator parallelism, resembling the rather simple, distributed approach, is achieved by executing distinctive operators of a query execution plan in a parallel or pipelined parallel fashion. In contrary the intra-operator parallelism is focusing on the parallel execution of a single operation among multiple processors by applying smart algorithms.

In the past a number of papers appeared covering this topic [NOD 93], [AME 90], [COP 88], [DEW 86], [DEW 90], [PIR 90], [STO 94], which proposed and analyzed parallel database algorithms for parallel database machines. [AMI 94] presents an adaptive, load-balancing parallel join algorithm implemented on a cluster of workstations. The algorithm efficiently adapts to use additional join processors when necessary, while maintaining a balanced load. [JIA 97] develops a parallel hash-based join algorithm using shared-memory multiprocessor computers as nodes of a networked cluster. [TAM 97] uses a hash-based join algorithm to compare the designed cluster-system with commercial parallel systems.

However, the referenced work touches specific implementations and algorithms only. From our point of view no in-depth theoretical and practical analysis and comparison for the most important parallel join algorithms on cluster architectures has been proposed until now. In this paper we will perform such an analysis and we will prove that cluster architectures can be a viable platform for parallel database systems.

The paper is organized as follows. In the next section we give a short survey of the underlying cluster framework. In the 3rd section we define the parallel join algorithms and develop a comprehensive cost model for evaluating their performance. In the 4th section we will present the performance analysis for the implemented algorithms and compare the practical results with the theoretical model. The paper finishes with conclusions.

Our intention is to give a case for parallel join operations on cluster architectures, but a focus on analyzing hardware characteristics of the underlying system is beyond the scope of this paper. So we are interested in the specifics of the algorithms and not of the machines.

5.2 Parallel framework

5.2.1 Cluster system architecture

The framework of our analysis is built on a very general model. The cluster architecture consists of 3 describing components,

- processing units (P_1, P_2,...P_k) with their main memories (M_1, M_2,...M_k)
- disks (D_1, D_2,...D_m)
- interconnect (primary, secondary)

A set of processing units, a global main memory and/or a set of disks build a *computational unit*. The connection between these components is via a primary, high-speed interconnect (e.g. a local bus, a system bus, etc.). A computational unit can be seen as a topological unit, in other words, all components reside on one board. As a special form we have computational units consisting of one processing unit, one main memory and one disk (Uniprocessor).

The cluster system (see Figure 5.1) consists of a number of such computational units (C_1, C_2, ...C_p) interconnected via a (secondary, slower) interconnect. This describes a system where the components reside on different boards (but can occupy one racket) or are spread across a network among different systems.

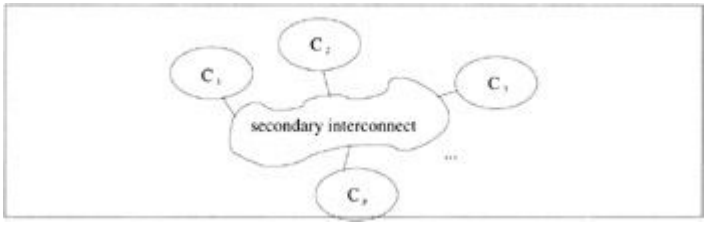


Figure 5.1: Architectural framework of a cluster system

5.2.2 Declustering

The general method in a parallel database system to increase the bandwidth of the I/O operations by reading and writing the multiple disks in parallel is *declustering*. This denotes the distribution of the tuples (or records, i.e. the basic data unit) of a relation among two or more disk drives, according to one or more attributes and/or a distribution criteria.

Three basic declustering schemes can be distinguished,

- range declustering (contiguous parts of a data set are stored on the same disk)
- round-robin declustering (tuples are spread consecutively on the disk)
- hash-declustering (location of a tuple is defined by a hash fuction)

The data tuples are grouped according to the declustering scheme with respect to the attribute values in one or a number [GHA 92] of the domains. The declustering scheme is expressed by a function d, which maps the tuples of a relation R to disk indices I. The following properties are valid:

$d : R \rightarrow I$, declustering function

$Ri = \{x | x \quad R \quad d(x) = i\}$, declustered sets R

with

$R = \quad R_i$ and $R_i \cap R_j = \quad$ if $i \neq j$

The disjoint property of the declustered sets R_i can be exploited by parallel algorithms based on the SPMD (single program, multiple data) paradigm. This means that multiple processors execute the same program, but each on a different set of tuples. In the database terminology this approach resembles the intra-operational parallelism.

A realistic assumption of our model is that the relations of the database system are too large to fit into the main memory of the processing units. Consequently, all operations have to be done external and the I/O costs are the dominant factor for the system performance.

5.3 Parallel joins

5.3.1 General information

Basically, the join operation 'merges' two relations *R* and *S* via two attributes (or attribute sets) *A* or *B* (respective relations *R* and *S*) responding to a certain join condition. The join attributes have to have the same domain. Two different types of join operators are distinguished, the equi-join and the theta-join. In the following only the equi-join is discussed.

In the literature [DAT 86] three different approaches for join algorithms are distinguished:

- sort merge,
- nested loop, and
- hash based join.

The *nested loop* technique basically compares each tuple (or page) of relation *R* with each tuple (or page) of relation *S*. Tuples which fulfil the join condition are merged.

The *sort merge* approach consists of two phases. In the first phase both relations *R* and *S* are sorted according to their attributes *A* and *B* and in the second both relations are sequentially scanned, compared and joined (i.e. merged).

The *hash-based* join builds hash tables on the relations in the first phase and probes in the second phase the tuples of one relation against the tuples of the other relation based on the hash information.

Basically, the parallel versions of the these approaches are realized in a conventional client-server scheme. The server stores both relations to join and distributes the declustered tuples among the available clients. The clients perform the specific join algorithm on their sub relations and send the sub results back to the server. The server collects the result tuples and stores the result relation.

In the following, the four join algorithms are presented with their specific characteristics. We develop a cost model for every join algorithm. We apply this cost model to estimate the costs of the different joins, which we use for the evaluation of the implementation.

5.3.2 General cost model

In the following we specify several parameters, which describe the characteristics of the model environment and build the basis for the derived cost functions[1]. Table 5.1 depicts the basic parameters of our cost model used in the following.

Table 5.1: Basic parameters of the cost model

m	number of tuples of relation R (inner relation)
n	number of tuples of relation S (outer relation)
p	number of processors
n_t_m	number of tuples per message: Data is exchanged between programs using messages. The size of a message is system dependent to achieve high communication performance.
b	bucket size (tuples per bucket): The bucket size defines the number of tuples per bucket, which is the basic unit of transfer between main memory and mass storage.
s	selectivity factor: The selectivity factor is a percentage of the product of m and n. It defines the number of result tuples and thus the costs of sending and writing the result data set.
l_f	loop_factor: Grace hash and Hybrid hash build buckets using a hash function. The data of the buckets has to be saved temporarily in a number of files. Due to the situation that the number of allowed open files is limited in practice, the creation of these temporary buckets has to be done in a certain number of loops (l_f)

Furthermore, we need a few derived terms which define the time to complete the specified action (see Table 5.2). Some of them are used in all four joins, the rest is only used for specific algorithms.

Table 5.2: Derived cost functions of the cost model

read	read one tuple from disk
write	write one tuple to disk
receive	receive one message
send	send one message
find_target	find the right target client find_target consists of applying a function to a tuple, which determines the right client and filling the tuple into a message.
hash	fill a tuple into a main memory hash table hash consists of applying a hash function on a tuple and saving the tuple in main memory.
probe	probe a main memory hash table with a tuple probe consists of applying a hash function on a tuple and producing a result tuple in main memory, if the keys match.
fill	fill a tuple into main memory fill stores a tuple in main memory.
compare	compare two tuples compare the keys of two tuples in main memory and build a result tuple if keys match.

The specific values of the basic parameters and the derived functions used in the theoretical model to calculate "real" numbers were profiled by a specific test program on the real hardware. The values are depicted in section 5.4.

The declustering of the data across the clients is equal for all four join algorithms and is therefore analyzed at the beginning. The server reads the two input relations [5.1],

$$server\_read = (m + n) * read \qquad [5.1]$$

calculates the respective target client using a declustering function with p as one of its parameters [5.2]

$$server\_compute = (m + n) * find\_target \qquad [5.2]$$

and sends the tuples (packed in messages) to the target client [5.3].

$$server\_send = \left(\frac{m}{n\_t\_m} + \frac{n}{n\_t\_m}\right) * send \qquad [5.3]$$

After sending the messages the server is in an idle-state. It waits for the results of the clients [5.4].

$$server\_receive = \frac{m * n * s}{n\_t\_m} * receive \qquad [5.4]$$

The tuples received are written to disk [5.5].

$$server\_write = m * n * s * write \qquad [5.5]$$

The total costs of the server are defined in [5.6] and are the sum of [5.1] to [5.5].

$$server\_cost = server\_read + server\_compute + server\_send + server\_receive + server\_write \qquad [5.6]$$

The costs of the server I/O-costs (read, write), message-costs (send, receive) and computational costs are obviously independent of the number of processors used.

5.3.3 Grace hash-join

A well known representative of a hash join algorithm is the Grace join algorithm [KIT 83]. It works in three phases. In the first phase, the algorithm declusters relation R (inner relation) into *N* disk buckets by hashing on the join attribute of each tuple in *R*. In phase 2, relation S (outer relation) is declustered into N buckets using the same hash function. In the final phase, the algorithm joins the respective matching buckets from relation R and S and builds the result relation.

The number of buckets, *N*, is chosen to be very large. This reduces the chance that any bucket of relation *R* will exceed the memory capacity of the processor used to actually effect the join (the buckets of the outer relation may exceed the memory capacity because they are only used for probing the bucket of the inner relation and need not to fit in main memory). In the case that the buckets are much smaller than main memory, several will be combined during the third phase to form more optimal sized join buckets (referred to as bucket tuning in [KIT 83]).

The Grace algorithm differs from the sort merge and simple-hash join algorithms in that data partitioning occurs at two different stages - during bucket forming and during bucket-joining. Parallelizing the algorithm thus must address both of these data partitioning stages. To ensure maximum utilization of available I/O bandwidth during the bucket-joining stage, each bucket is partitioned across all available disk drives. A *partitioning split table* is used for this task. To join the i^{th} bucket in *R*

with the i^{th} bucket of *S*, the tuples from the i^{th} bucket in *R* are distributed to the available joining processors using a *joining split table* (which will contain one entry for each processor used to effect the join). As tuples arrive at a site they are stored in in-memory hash tables. Tuples from bucket *I* of relation *S* are then distributed using the same joining split table and as tuples arrive at a processor they probe the hash table for matches. The bucket-forming phase is completely separate from the joining phase under the Grace join algorithm. This separation of phases forces the Grace algorithm to write the buckets of both joining relations back to disk before beginning the join stage of the algorithm.

5.3.3.1 Cost model

The costs for the clients start with receiving the tuples of the inner and outer relation from the server. Every client gets only $\frac{m}{p}$ tuples of the inner relation R and $\frac{n}{p}$ tuples of the outer relation S. The costs for receiving are described by [5.7].

$$client\_receive = \frac{\frac{m}{p}}{n\_t\_m} * receive + \frac{\frac{n}{p}}{n\_t\_m} * receive \qquad [5.7]$$

Afterwards the received tuples are written to the local disk, which is [5.8].

$$build\_temp\_files = \frac{m}{p} * write + \frac{n}{p} * write \qquad [5.8]$$

In l_f loops buckets are built using a hash table. We need l_f loops because there is a restriction to the number of possible open files within a system. The formula for these costs is given by [5.9].

$$build\_buckets = (\frac{m}{p} * l\_f + \frac{n}{p} * l\_f) * (read + hash + write) \qquad [5.9]$$

In the third phase the buckets of relation *R* are read and an in-memory hash table is filled with the tuples of the buckets. The costs are described by [5.10].

$$build\_hash = \frac{m}{p} * (read + hash) \qquad [5.10]$$

The hash table is proved for every tuple of *S*. If the tuples match, a result tuple is created [5.11].

$$probe\_hash = \frac{n}{p} * (read + probe) \qquad [5.11]$$

Finally the client has to write the result tuples back to the server [5.12]. The complete costs of the client [5.13] are the sum of [5.6] to [5.11]. The total cost for the Grace hash join [5.13] is the sum of the cost of the server and [5.12].

$$send\_result = (\frac{m}{p} * \frac{n}{p}) * \frac{s}{n\_t\_m} * send \qquad [5.12]$$

$$client\_grace\_cost = client\_receive + build\_temp\_files + build\_buckets + build\_hash + probe\_hash + send\_result \qquad [5.13]$$

$$grace\_cost = server\_cost + client\_grace\_cost \qquad [5.14]$$

The theoretical speed-up of the Grace hash-join algorithm based on our model is shown in Figure 5.2. Independently of the number of input tuples we can calculate a significant speedup. Thus we see a perfect linear behavior of the algorithm. Please note that the x axis is logarithmic.

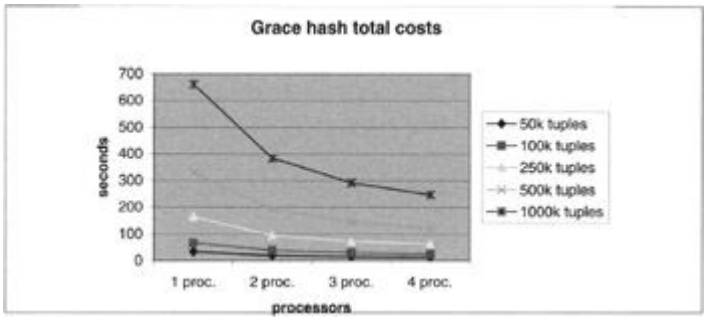


Figure 5.2: Theoretical Grace hash-join speed-up

Figure 5.2 shows the behavior of the algorithm with increasing workload[2]. It can be seen that duplication of the number of input tuples leads to a duplication of costs independent of the number of processors (1 to 4). Additionally, we see again the improvement of performance when using more processors, which justifies our first result.

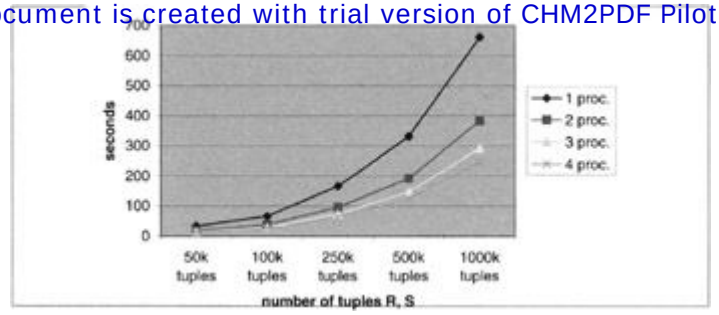


Figure 5.3: Theoretical Grace hash-join scale-up

It is interesting to check the percentage of I/O-costs, message costs and computational costs in relation to the total client costs. The model shows a steady distribution of the different parts of costs while changing the number input tuples. This could be expected because Grace hash is of linear order in n.

5.3.4 Hybrid hash-join

Another hash join algorithm is the centralized Hybrid hash-join [[DEW 84]], which operates similarly to the Grace hash-join in three phases. In the first phase, the algorithm uses a hash function to partition the inner relation *R* into N buckets. The tuples of the first bucket are used to build an in-memory hash table while the remaining *N* – 1 buckets are stored in temporary files. A good hash function distributes the data uniformly and produces just enough buckets to ensure that each bucket of tuples will be small enough to fit entirely in main memory. During the second phase, relation *S* is partitioned using the hash function from the first step. Again, the last *N* – 1 buckets are stored in temporary files while the tuples of the first bucket are used to immediately probe the in-memory hash table built during the first phase. During the third phase, the algorithm joins the remaining *N* – 1 buckets from relation *R* with their respective buckets from relation *S*. The join is thus broken up into a series of smaller joins. The third phase of the Hybrid hash-join algorithm works in the same way as the Grace hash-join except the first bucket is already joined.

The parallel version of the Hybrid hash-join algorithm is similar to the centralized version described above. The relations *R* and *S* are declustered. On each client the tuples of the joining relations are separated into *N* buckets using a *partitioning split table*. The number of buckets is chosen such that the tuples corresponding to each logical bucket will fit in the aggregate memory of the joining processors. Furthermore, the partitioning of the inner relation *R* into buckets is overlapped with the insertion of tuples from the first bucket of *R* into main-memory. In addition, the partitioning of the outer relation *S* into buckets is overlapped with the joining of the first bucket of *S* with the first bucket of *R*. If overflow occurs by building the first (main-memory) bucket of *R*, the corresponding first bucket of *S* must be stored on disk.

A very early performance evaluation of the Hybrid hash-join in a shared-nothing multiprocessor environment is presented in [SCH 89].

5.3.4.1 Cost model

The cost model for Hybrid hash is nearly identical to the cost model of Grace hash. There is only one difference in the costs of building buckets. Hybrid hash stores the first bucket directly into main memory. So we have to change [5.9] to [5.15] as follows:

$$build\_buckets\_hybrid = ((\frac{m}{p} * l\_f + \frac{n}{p} * l\_f) - b) * (read + hash + write) \quad [5.15]$$

The complete costs of the client is shown in [5.16]. The total cost for the Hybrid hash join [5.17] is the sum of the cost of the server and [5.15].

$$client\_hybrid\_cost = client\_receive + build\_temp\_files + build\_buckets\_hybrid + build\_hash + probe\_hash + send\_result \quad [5.16]$$

$$hybrid\_cost = server\_cost + client\_hybrid\_cost \quad [5.17]$$

The improvement of performance using more processors is similar to the situation for Grace hash-join and is shown in Figure 5.4. Handling a growing number of input tuples makes the special treatment of the first bucket more and more unimportant. The duplication of the number of input tuples leads to a duplication of costs independent of the number of processors as seen with the Grace hash-join. The percentage of I/O-costs, message costs and computational costs in relation to the total client costs stay steady while changing the number input tuples. A change of the number of processors does not change the percentage of I/O-costs, message costs and computational costs in relation to the total client costs.

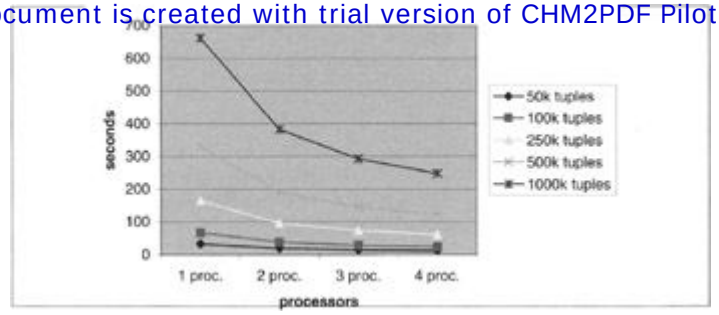


Figure 5.4: Theoretical Hybrid hash-join speed-up

5.3.5 Nested loop join

The nested loop algorithm is the most simplest approach to join two relations. Basically each tuple of one relation is probed to each tuple of the other relation. However, its simple layout and thus low constant cost factors (overhead) makes it quite attractive in database systems for specific situations (e.g. one relation very small).

A parallel approach of the nested loop join partitions first the inner relation *R* across the clients. At the clients the tuples are stored in a temporary file. Second, the outer relation *S* is distributed among the disks using the same hash function as in the first step. In the third phase both relations are joined. Step by step the main memory is filled with tuples of relation *R*. With every step the complete relation *S* is read and every tuple of *S* probes the content (tuples of *R*) in the main-memory. In case of a match result tuples are built.

5.3.5.1 Cost model

The costs for receiving the tuples are the same as in the Grace hash join and Hybrid hash join. Therefore we can use [5.7]. Then the received tuples are written to the local disk [5.8].

The filling of the memory and the probing of the tuples is depicted by [5.18] and [5.19] respectively.

$$build\_mem = \frac{m}{p} * (read + fill) \qquad [5.18]$$

$$probe\_mem = (\frac{\frac{m}{p}}{b}) * (\frac{n}{p} * read + \frac{n}{p} * b * compare) \qquad [5.19]$$

At last the algorithm sends back the join results. The formula is the same as for both hash joins. So we can use [5.12]. The complete costs of the client [5.13] are the sum of [5.7],[5.8],[5.18],[5.19] and [5.12]. The total cost for the nested loop join [5.21] is the sum of the cost of the server and [5.20].

$$client\_nested\_cost = client\_receive + build\_temp + build\_mem + probe\_mem + send\_result \qquad [5.20]$$

$$nested\_cost = server\_cost + client\_nested\_cost \qquad [5.21]$$

Figure 5.5 shows the performance of the nested loop join using different numbers of processors and a different numbers of input tuples. The algorithm improves using more processors but is generally inferior. So we cannot show the performance for a higher number of input tuples in Figure 5.5.

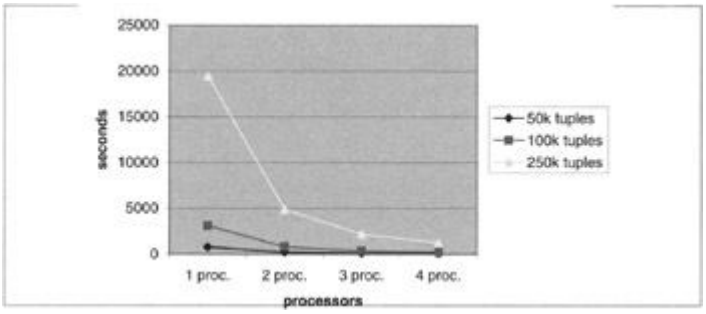


Figure 5.5: Theoretical nested loop join speed-up

Next we want to know if the costs per tuple change when the number of input tuples increase. We expect that the costs per tuple rise because the main parts of nested loop join develop with $O(m + n * m)$ where $|n| < |m|$, so simply stated $O((m + n)^2)$. The results can be seen in Figure 5.6 for 1 to 4 clients.

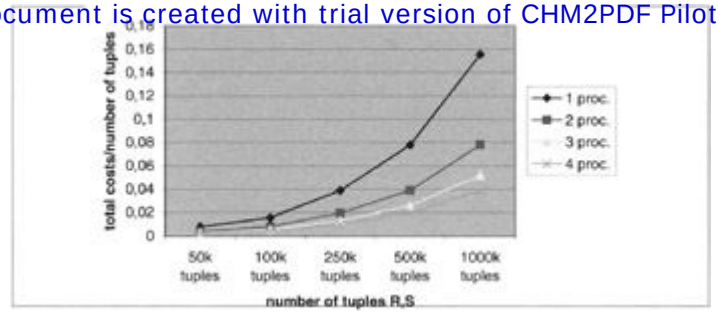


Figure 5.6: Theoretical nested loop join cost per tuple

Further, we checked the percentage of I/O-costs, message costs and computational costs in relation to the total client costs. It shows a nearly even distribution of the different parts of costs while changing the number of input tuples. At least we want to know if there is a change in the percentage of I/O-costs, message costs and computational costs in relation to the total client costs when changing the number of used processors. Again, it shows an even distribution of the different parts of costs. Most of the execution time is used for computing.

5.3.6 Sort merge join

The parallel version of the sort merge join algorithm is a straightforward adaption of the traditional single processor version of the algorithm [COR 96]. The inner relation *R* is first partitioned using a *split table* (range declustering). A function is applied to the join attribute of each tuple to determine the appropriate disk site. As the tuples arrive at a client they are gathered in buckets. The buckets are sorted in ascending order of the join attribute and written to two temporary files of equal size. Every client uses binary sort merge to sort its part of relation *R*. Parallel binary merge sort is described in [BIT 83]. An analysis and evaluation of parallel binary merge sort is given in [SCH 96]. In our algorithm we use only the *suboptimal* and *optimal* part of the sort algorithm. The suboptimal phase (see Figure 5.7) merges pairs of longer and longer runs (i.e. ordered sequences of pages). In every step the length of the runs is twice as large as in the preceding run. At the beginning each processor reads two sorted pages, merges them into a run of 2 pages length and writes it back to the disk. This is repeated until all buckets are read and merged into 2-buckets-runs. If all buckets are merged, the *suboptimal phase* continues with merging two 2-page-runs to a sorted 4-page-run. This continues until all 2-page-runs are merged. The phase ends when the two temporary files are sorted. At the end of the *suboptimal phase* exist on each node 2 sorted temporary files. During the following *optimal phase* each processor merges the 2 temporary files (see Figure 5.8).

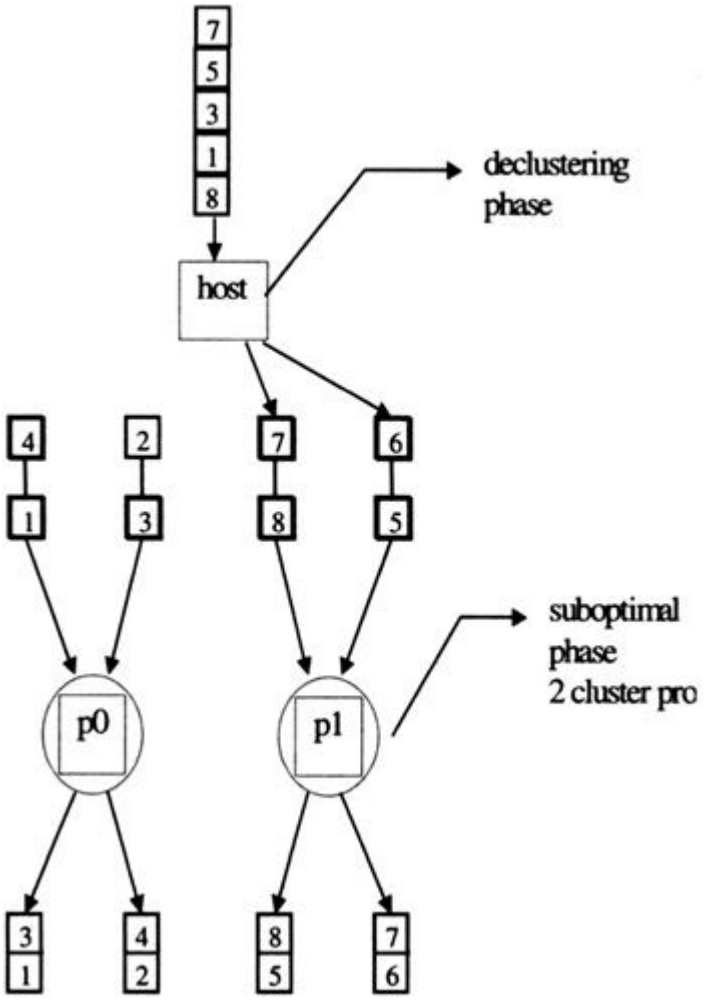


Figure 5.7: Suboptimal sort merge join phase

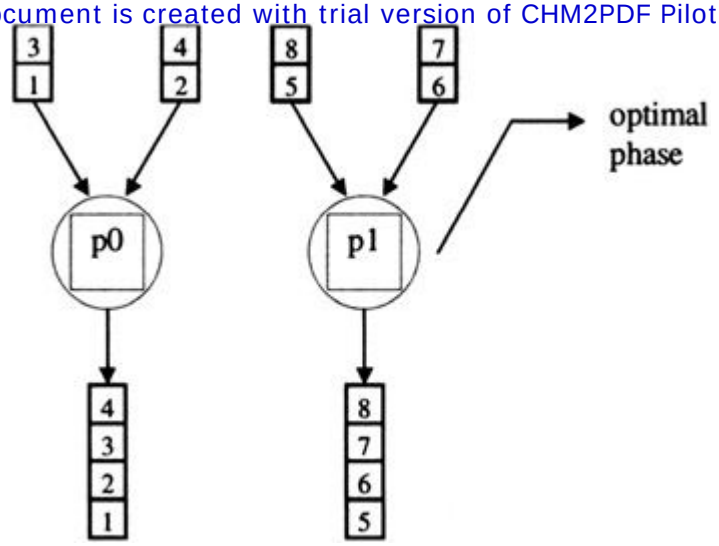


Figure 5.8: Optimal sort merge join phase

In a second phase the outer relation *S* is partitioned using the same *split table*. Every client receives its tuples of relation *S*, builds and sorts buckets and sorts the temporary files of relation *S*using binary sort merge. In a third phase the sorted temporary files of *R* and *S* are merged and result tuples are built.

5.3.6.1 Cost model

The costs for receiving the tuples are the same as in Grace hash and Hybrid hash. Therefore we can use [5.7]

In the following step every bucket has to be sorted. Because we use nested loop sort for sorting the buckets the formula is

$$sort\_bucket = (\frac{m}{p*b} + \frac{n}{p*b}) * b^2 * compare \qquad [5.22]$$

After sorting the buckets are written to the local disk. The formula is given in [5.8].

In the suboptimal phase and the optimal phase of binary merge we have to sort the two input relation.

$$sort\_R = (\frac{m}{p} * log\frac{m}{p}) * (read + compare + write) \qquad [5.23]$$

$$sort\_S = (\frac{n}{p} * log\frac{n}{p}) * (read + compare + write) \qquad [5.24]$$

Next the sorted input relations have to be merged and result tuples have to be built, if necessary.

$$merge = (\frac{m}{p} + \frac{n}{p}) * (read + compare) \qquad [5.25]$$

At last the algorithm sends back the join results. The formula is the same as for the other joins. So we can use [5.12]. The complete costs of the client are given in [5.26]. The total cost for the sort merge join [5.27] is the sum of the cost of the server and [5.26].

$$client\_sort\_cost = client\_receive + build\_temp + sort\_bucket + sort\_R + sort\_S + merge + send\_result \qquad [5.26]$$

$$nested\_cost = server\_cost + client\_sort\_cost \qquad [5.27]$$

Figure 5.9 shows the performance of the sort merge join using a different number of processors and a different number of input tuples. The algorithm improves using more processors showing a speedup between the two hash joins and the nested loop join.

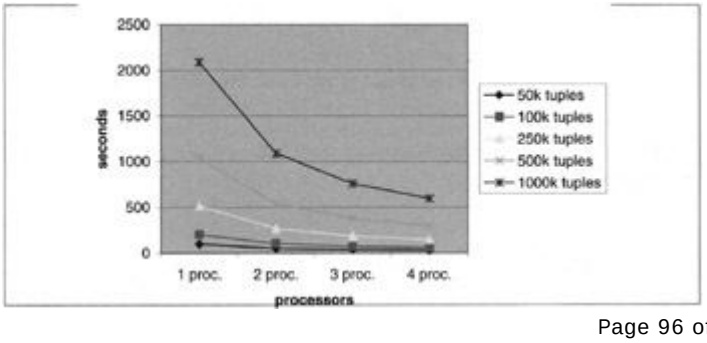


Figure 5.9: Theoretical sort merge join speed-up

The costs per tuple change only slightly when the number of input tuple increases. We expect that the costs per tuple rise because the main parts of sort merge develop with $O((m + n) * log(m + n))$. The results can be seen in Figure 5.10 for 1 to 4 clients.

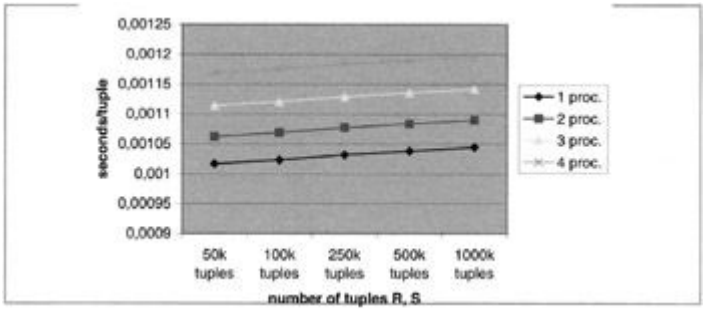


Figure 5.10: Theoretical sort merge join cost per tuple

In another test we control the percentage of I/O-costs, message costs and computational costs in relation to the total client costs.

At least we want to know if there is a change in the percentage of I/O-costs, message costs and computational costs in relation to the total client costs when changing the number of processors used. It shows again an even distribution of the different parts of costs.

[1]For the heuristic analysis in section 5.4 we used a portable benchmark to check the real, system dependent values for these parameters.

[2]Please note the quasi-logarithmic scale of the x axis.

5.4 Performance analysis

For the practical performance analysis all algorithms were realized as described in the preceding section. One of the nodes acted as server, starting the operations, distributing the workload and collecting the result, and 4 client nodes, processing the distributed workload. At the beginning of each test run the relations *R* (inner relation) and *S* (outer relation) reside on a server. In the tests we used an integer variable as join attribute.

Test-bed for our analysis was an off-the-shelf "el-cheapo" PC cluster consisting of 5 single processor nodes (computational units) running the Linux operating system. The algorithms were realized with the C language and PVM [SUN 90] as communication library. We use *pvm_spawn* to start the client tasks. The modules use *pvm_mytid, pvm_parent, pvm_exit* to control themselves. The connection between the server and the client tasks is realized by using *pvm_send, pvm_recv*. The handling of the single messages (tuples), i.e. compressing and decompressing uses *pvm_initsend, pvm_pkint, pvm_pkstr, pvm_bufinfo, pvm_upkint, pvm_upkstr*. To handle errors we use *pvm_perror*.

The specific values of the parameters implemented in our tests are depicted in Table 5.3. Compare the parameters with Table 5.2 of the theoretical analysis.

Table 5.3: Specific values of the basic parameters

m	50000,100000,250000,500000,1000000
n	50000,100000,250000,500000,1000000
p	1,2,3,4
n_t_m	100
b	1000
s	1/m
l_f	10

We used a test module to determine the values of the basic parameters and the derived functions (measured in seconds) in our cost model. The values have been determined by using the C-gettime-function by a profiling module. The results can be seen in Table 5.4.

Table 5.4: Values for derived functions of the cost model

read	0,0000105 seconds
write	0,00001 seconds
receive	0,0025 seconds
send	0,0025 seconds
find_target	0,000005 seconds
Hash	0,00001 seconds
probe	0,00001 seconds
fill	0,0000008 seconds
compare	0,0000008 seconds

The following figures give the real execution times for the 4 algorithms, Grace hash join, Hybrid hash join, sort merge join, and nested loop join, in Figures 5.11, 5.12, 5.13, and 5.14 respectively. For the nested loop algorithm only the execution time for 50000 elements is shown, due to exhaustive run times. All values given are the averages of at least 20 runs.

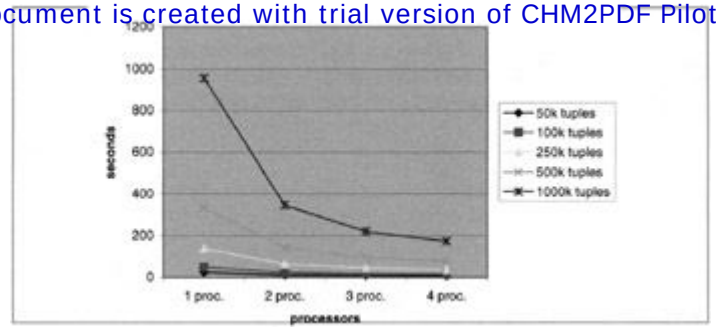


Figure 5.11: Real Grace hash join speed-up

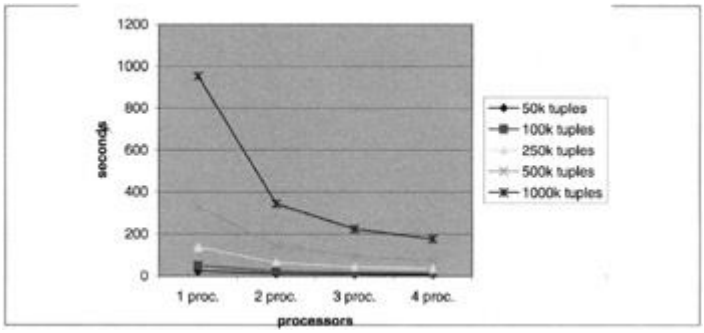


Figure 5.12: Real Hybrid hash join speed-up

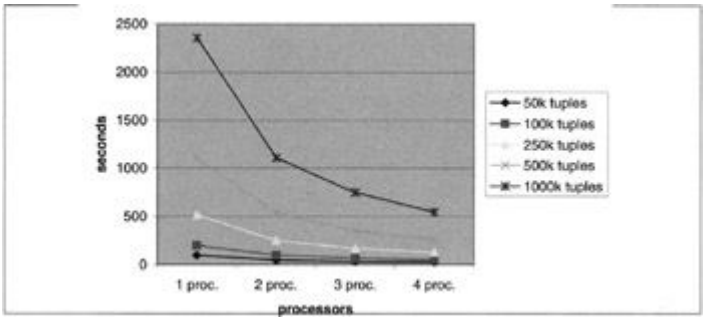


Figure 5.13: Real sort merge join speed-up

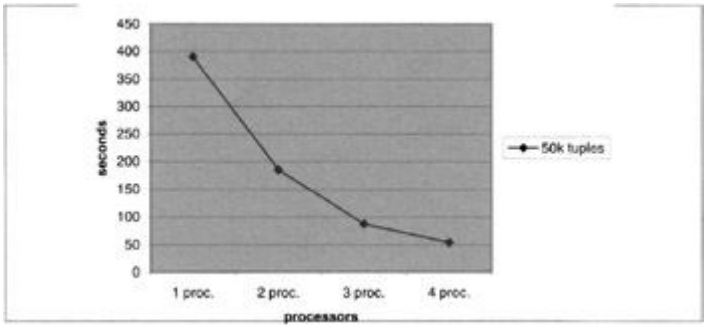


Figure 5.14: Real nested loop join speed-up

The real values correspond the theoretical values amazingly well. Not only the runtime behavior for increasing workloads and processing nodes fit, but also the real execution values match the ones calculated by the model.

5.5 Conclusions

Summing up we gave a case for the usage of cluster systems as a basis architecture for parallel database systems. With the development of faster and cheaper network interfaces clusters can deliver an un-beatable price/performance ratio for many former pure supercomputer applications, as in our case the administration and manipulation of very large database systems.

5.6 Acknowledgements

First we would like to express our thanks to Thomas Fuerle, who was always helpful in the administration of the cluster and the programming of the test suite. Further we thank the anonymous reviewers for the valuable comments, which helped to improve the paper.

5.7 Bibliography

[ABD 98] Abdelguerfi M., Wong K.-F., *Parallel Database Techniques*, IEEE Computer Society Press, 1998.

[AME 90] America P., *"Parallel Database Systems"*, *Proc. PRISMA Workshop*, Springer Verlag, 1990.

[AMI 94] Amin M. B., Schneider D. A., Singh V., *"An Adaptive, Load Balancing Parallel Join Algorithm"*, *Sixth International Conference on Management of Data (COMAD'94)*, Bangalore, India, 1994.

[BIT 83] Bitton D., Boral H., Dewitt D., Wilkinson W., *"Parallel Algorithms for the Execution of Relational Operations"*, *ACM Trans. Database Systems*, vol. 8, num. 3, 1983, p. 324-353.

[COP 88] Copeland G., Alexander W., Boughter E., Keller T., *"Data Placement in Bubba"*, *Proc. Of the ACM-SIGMOD Int. Conf. On Management of Data*, ACM, 1988.

[COR 96] Cormen T. H., Leiserson C. E., Rivest R. L., *Introduction to Algorithms*, MIT Press and McGraw-Hill, 1996.

[DAT 86] Date C., *An Introduction to Database Systems*, Addison-Wesley, 1986.

[DEW 84] Dewitt D. J., Katz R. H., Olken F., Shapiro L. D., Stonebraker M. R., Wood D., *"Implementation techniques for main memory database systems"*, *SIGMOD Record (ACM Special Interest Group on Management of Data)*, vol. 14, num. 2, 1984, p. 1-8.

[DEW 86] Dewitt D., Gerber R., Graefe G., Heytens M., Kumar K., Muralikrishna M., *"GAMMA- a High Performance Dataflow Database Machine"*, *Proc. Of the 12th Conf. On Very Large Data Bases*, 1986.

[DEW 90] Dewitt D., Gray J., *"Parallel Database Systems: The Future of Database Processing or a Passing Fad?"*, *ACM-SIGMOD record*, vol. 19, num. 4, 1990.

[GHA 92] Ghandeharizadeh S., Dewitt D., Qureshi W., *"A Performance Analysis of Alternative Multi-Attribute Declustering Strategies"*, *Proc. Of the 1992 ACM-SIGMOD Int. Conf. On Management of Data*, ACM, 1992.

[JIA 97] Jiang Y., Makinouchi A., *"A Parallel Hash-Based Join Algorithm for a Networked Cluster of Multiprocessor Nodes"*, *Proceedings of the COMPSAC '97 - 21st International Computer Software and Applications Conference*. 1997.

[KIT 83] Kitsuregawa M., Tanaka H., Moto-Oka T., *"Application of Hash to Data Base Machine and Its Architecture"*, *New Generation Computing*, vol. 1, num. 1, 1983.

[NOD 93] Nodine M. H., Vitter J. S., *"Optimal Deterministic Sorting on Parallel Disks"*, *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '93)*, Velen, Germany, 1993, p. 120-129.

[PIR 90] Pirahesh H., Mohan C., Cheng J., Liu T., Selinger P., *"Parallelism in Relational Database Systems: Architectural Issues and Design Approaches"*, *Proc. Of the IEEE Conf. On Distributed and Parallel Database Systems*, IEEE Computer Society Press, 1990.

[SCH 89] Schneider D., Dewitt D., *"A Performance Evaluation of Four Parallel Join Algorithms on a Shared-Nothing Multiprocessor Environment"*, *Proc. Of the 1989 ACM-SIGMOD*, ACM, 1989.

[SCH 96] Schikuta E., Kirkovits P., *"Analysis and evaluation of sorting for parallel database systems"*, *Proc. Euromicro 96, Workshop on Parallel and Distributed Processing*, Braga, Portugal, 1996, IEEE Computer Society Press, p. 258-265.

[SEG 00] Segal B., *"Grid Computing: The European Data Project"*, *IEEE Nuclear Science Symposium and Medical Imaging Conference*, Lyon, France, 2000, IEEE Computer Society Press.

[STO 94] Stonebraker M., Aoki P., Devine R., Litwin W., Olson M., *"Mariposa: A New Architecture for Distributed Data"*, *Proc. Of the Int. Conf. On Data Engineering*, IEEE Computer Society Press, 1994.

[SUN 90] Sunderam V. S., *"PVM: A Framework for Parallel Distributed Computing"*, *Concurrency: Practice and Experience*, vol. 2, num. 4, 1990, p. 315-339.

[TAM 97] Tamura T., Oguchi M., Kitsuregawa M., *"Parallel Database Processing on a 100 Node PC Cluster"*, *Proc. of the Supercomputing 97*, IEEE Computer Society Press, 1997.

Chapter 6: Server-side Scheduling in Cluster Parallel I/O Systems

Robert B. Ross, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, USA
Walter B. Ligon III, Holcombe Department of Electrical and Computer Engineering, Clemson University, Clemson, SC 29634,
USA

6.1 Introduction

Parallel I/O has become a necessity in the face of performance improvements in other areas of computing systems. Studies have shown that peak performance is infrequently realized, and work in parallel I/O optimization strives to achieve peak performance for applications. In this paper we revisit one area of performance optimization in parallel I/O, that of server-side scheduling of service. With the wide variety of systems and workloads seen today, multiple server-side scheduling algorithms are necessary to match potential workloads. We show through experimentation that performance gains can be seen in practice through the use of alternative scheduling algorithms but that no single algorithm provides the best performance across the board. We discuss the potential for automatic matching of server-side scheduling algorithms to workloads in real time.

Performance improvements in computing technology have vastly outpaced improvements in storage technology. This trend has led to the adoption of parallel I/O systems as a solution. By combining large numbers of storage devices and providing the system software to use them in concert, parallel I/O has extended the range of problems that may be solved on high-performance computing platforms. Workload studies indicate, however, that peak performance is rarely attained from these coordinated storage devices.

In order to address this situation, a collection of techniques for more efficiently utilizing these resources has been researched and developed. The collection includes traditional I/O enhancements such as prefetching and caching as well as novel approaches to organizing storage access and data transfer in parallel systems. The majority of these approaches were tailored for the environment in which they were originally applied, that of commercial supercomputers. In commercial parallel machines the network typically has much lower latency and much higher throughput than the storage system, so accounting for a bottleneck on the storage side has been the goal of much parallel I/O work.

More recently, new parallel computing platforms have emerged, including PC clusters such as Beowulf computers [RID 97]. Beowulfs are constructed from commodity components and commonly include fast ethernet networks and IDE disks. These components are of roughly the same order of magnitude of performance, so no one subsystem stands out as being a consistent bottleneck. The widespread adoption of clusters as a high-performance computing platform has seen the same I/O techniques developed for commercial supercomputers applied again, but no work has thus far examined the viability of these approaches in this new arena.

In this work we focus on the application of server-side scheduling algorithms in parallel I/O workloads on cluster systems. First, we cover previous work related to scheduling of server operations. Then, we describe PVFS, a parallel file system developed for Linux clusters, and we describe the set of scheduling algorithms we have implemented that work with PVFS. Next, we describe the set of workloads used in our tests to provide a variety of access patterns and resource demands. Finally, we examine the results of our experiments. From these experiments we note that the application of scheduling on the server side does have noticeable effects on performance and that a correlation between resource demand and optimal algorithm choice can be seen. We describe how the results of this work might be used to implement a scheduling system that can dynamically apply scheduling algorithms, in real time, based on workload information.

6.2 Server-side scheduling

The networks in use during the majority of early parallel I/O work were of much higher performance than the storage subsystems provided in the same systems. For example, the iPSC/860 system at NAS [NIT 92] had a total of ten I/O nodes using SCSI disks with a peak of 1 Mbyte/sec providing storage for the CFS parallel file system. The hypercube-based network provided 2.8 Mbytes/sec connections between processors. At least in part because of this environment, early techniques for optimizing access tended to focus on optimizing disk performance. The first four techniques covered in this section - data sieving, two-phase I/O, disk-directed I/O, and server-directed I/O - were all originally applied in or designed for these types of systems. The last technique discussed, stream-based I/O, was designed instead with cluster systems in mind.

6.2.1 Data sieving

Data sieving is a technique for efficiently accessing noncontiguous regions of data in files when noncontiguous accesses are not provided as a file system primitive [CHO 94]. The technique is presented here because it is a building block for two-phase access, which is discussed in the next subsection.

Workload studies on a number of platforms have shown that noncontiguous accesses commonly occur in parallel I/O workloads [KOT 95]. The naive approach to accessing noncontiguous regions is to use a separate I/O call for each contiguous region in the file. This results in a large number of I/O operations, each of which is often for a very small amount of data. The added network cost of performing an I/O operation across the network, as in parallel I/O systems, is often high because of latency. Thus, this naive approach typically performs very poorly because of the overhead of multiple operations. In the data sieving technique, a number of noncontiguous regions are accessed by reading a block of data containing all of the regions, including the unwanted data between them (called "holes"). Figure 6.1 shows an example of how data sieving might access a number of noncontiguous regions by reading a single block. The regions of interest are then extracted from this large block by the client. This technique has the advantage of a single I/O call, but additional data is read from the disk and passed across the network. When data sieving was implemented on an Intel Touchstone Delta, the reduction of operations outweighed the added data transfer for a large percentage of accesses.

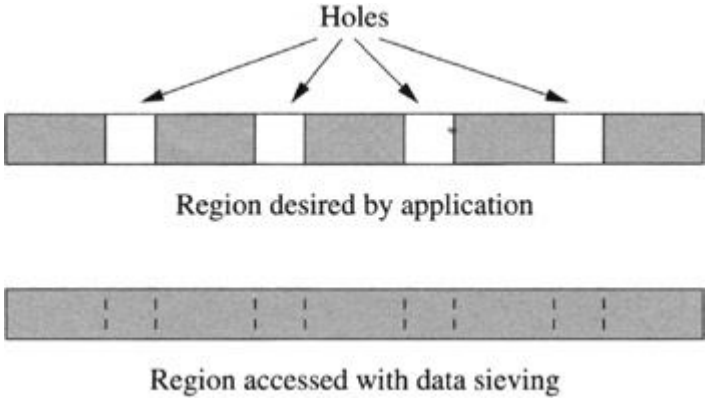


Figure 6.1: Data sieving example

This technique can also benefit systems with high latency networks in that it reduces the number of requests, for which there is often significant startup time. However, the percentage of data transferred that is desired must be high for this to pay off. A more appropriate technique for reducing the overhead of multiple requests in network-bound systems is the use of more descriptive requests. These allow the I/O server either to perform noncontiguous accesses, if the capability is available, or to perform this sieving on the server side, reducing network traffic. However, most I/O systems support only contiguous accesses.

6.2.2 Two-phase I/O

The *two-phase access* strategy is described in [BOR 93]. This strategy attempts to avoid the performance penalties often incurred when directly mapping from the distribution of data on disks to the distribution in processor memories. Data is first read from disk, in the arrangement it is stored in on disk, by a subset of processors. The data is then redistributed to the processors in the final processor-memory distribution. The strategy was tested on the Intel Touchstone Delta using the Concurrent File System (CFS). The 512-processor Delta has a limited number of I/O nodes (32) and substantial network bandwidth between processors, which are arranged in a mesh topology.

In the first phase of access, the number of processors involved is chosen to match the I/O nodes. Each chosen processor typically requests all the data needed from a single disk and uses data sieving to reduce the number of requests. In the second phase, the processors that previously read data from the disks calculate the final destination for each block of data and perform the necessary transfers. This strategy takes advantage of the additional bandwidth between compute processors

to complete the I/O process more quickly.

In order for this technique to be of use, compute processes must communicate and organize the transfer, which means that collective I/O must be available. The collective component of two-phase access can be implemented above the file system layer (i.e., on top of a file system that does not support collective accesses). Possibly the most popular implementation of two-phase I/O at this time is in the ROMIO MPIIO implementation [ROM], which implements two-phase accesses on top of a variety of parallel file systems with only independent access primitives.

This technique indirectly affects the behavior of I/O servers by altering the request pattern from many small accesses into single large accesses per server. In effect the server is forced into a mode where it is sequentially accessing a single large region (assuming the server returns the bytes from the request in order). It also constrains the server to servicing requests for a single client, since only one client makes a request. When disk is the bottleneck, this technique is often a win. Additionally, research has shown that some I/O systems perform best when the number of simultaneous accesses is limited [KRY 93, NIT 92]. These systems in particular benefit from this approach.

6.2.3 Disk-directed I/O

Disk-directed I/O (DDIO) is a combination of a number of other techniques for data transfer in parallel I/O systems [KOT 97]. DDIO was developed after both the data sieving and two-phase techniques, and it relies on both noncontiguous and collective I/O primitives in the file system. Additionally, the I/O servers must be able to map file locations into disk block positions and must be capable of reasonably predicting the optimal disk access pattern. The disk-directed technique uses the information passed to it about the data requested in the collective request to determine a list of physical blocks to retrieve from the disk. It sorts these blocks into some optimal access ordering and uses double buffering to overlap disk and network I/O, sending data directly to the final destination.

From the server scheduling point of view the disk-directed approach is superior to the two-phase technique in that it passes a great deal more information on the total access along to the server. This allows the server to determine the access ordering using both information on what is being accessed and information on where that data is located on disk. Furthermore, the disk-directed approach gives the server the opportunity to schedule access across multiple network connections, as data is moved directly from the server to the appropriate clients.

As far as contributions to a well-rounded I/O transfer method, DDIO has a number of features to offer. First, it uses noncontiguous requests, generally resulting in fewer, larger packets. Second, it promotes the use of an ordering scheme for optimization on the server side. While it might not always make sense to optimize for disk access, and a static scheme such as the one used in their examples might not help in a complex system, it does make sense to have a system capable of determining the cost of transferring particular packets and ordering transfers accordingly. Unfortunately, most parallel I/O systems do not meet the requirements for implementing DDIO. Server-directed I/O relaxes these requirements.

6.2.4 Server-directed I/O

A derivative of disk-directed I/O, called server-directed I/O, was proposed and implemented in the PANDA library [SEA 95]. This technique uses a high-level multidimensional data set interface, performs array chunking, and uses disk-directed techniques at the logical, or file, level. Instead of determining physical block locations, Seamons and his colleagues use logical file offsets to determine their optimal ordering. File data was stored on underlying local file systems, and block arrangement information was unavailable. Nevertheless, the developers found that they were able to use almost the full capacity of the disk subsystems in their test system for a range of array sizes and numbers of nodes.

6.2.5 Stream-based I/O

Stream-based I/O (SBIO) attempts to address network bottlenecks in parallel I/O systems [LIG 96]. The SBIO technique was developed as part of the Parallel Virtual File System (PVFS) project [LIG 96], which is described in Section 6.3. With SBIO, this concept of combining small accesses into more efficient, large ones is applied to data transfer over the network. Data being moved between clients and servers is considered to be a stream of bytes regardless of the location of data bytes within a file. This technique is similar to message coalescing in interprocessor communication. The streams are packetized by underlying network protocols (e.g., TCP) for movement across the network. Control messages are placed only at the beginning and end of the data stream in order to minimize their effects on packetization. This is accomplished by calculating the stream data ordering on both client and server.

SBIO is strictly a technique for optimizing network traffic. When coupled with a server that focuses on the network (almost "network-directed I/O"), peak performance can be maintained for a variety of workloads, particularly when network performance lags behind disk performance or when most data on I/O servers is cached.

6.3 PVFS design

One area in which commercial machines still maintain great advantage is that of parallel file systems. A production-quality high-performance parallel file system has not been available for Linux clusters, and without such a file system, Linux clusters cannot be used for large I/O-intensive parallel applications. To fill this need, we have developed a parallel file system for Linux clusters called the Parallel Virtual File System (PVFS). PVFS is being used by a number of groups, including those at Argonne National Laboratory and the NASA Goddard Space Flight Center. Other researchers are using the PVFS system as a research tool [TAK 99].

Details on PVFS can be found in [CAR 00]. The rest of this section focuses on the components of PVFS that are important to this study, namely, how PVFS manages data storage, how PVFS processes requests for data, and how this request processing might be modified to study server-side scheduling algorithms.

6.3.1 PVFS metadata

PVFS files are striped across a set of I/O nodes in round-robin fashion. The specifics of a given file distribution are described with three metadata parameters: starting I/O node number (*base*), number of I/O nodes (*pcount*), and strip size (*ssize*). These parameters, together with an ordering of the I/O nodes for the file system, allow the file distribution to be completely specified.

An example of some of the metadata fields for a file `/pvfs/foo` is given in Figure 6.2. The *pcount* field specifies that the data is spread across three I/O nodes, *base* specifies that the first (or base) I/O node is node 2, and *ssize* specifies that the strip size-the unit by which the file is divided among the I/O nodes-is 64 Kbytes. The application can set these parameters when the file is created, or, if not specified, PVFS will use a default set of values.

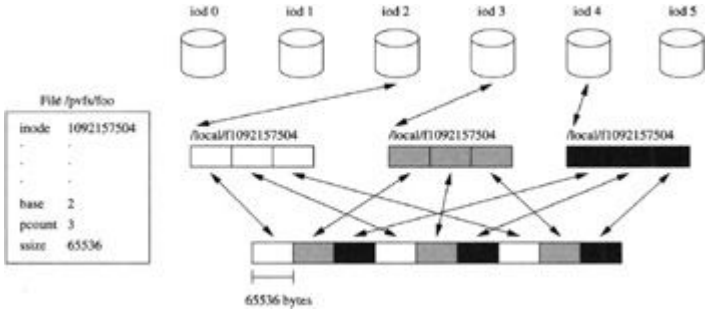


Figure 6.2: Example of metadata and file distribution

This file metadata, including locations of I/O nodes, is obtained by the application from the PVFS system when a file is opened. This information allows applications to communicate directly with I/O nodes when file data is accessed.

6.3.2 I/O daemons and data storage

An ordered set of I/O daemons (iods) runs on the I/O nodes in the cluster. The I/O nodes are specified by the administrator when the file system is installed. These daemons are responsible for using the local disks on each I/O node for storing data for PVFS files, and they do so by using a local file system to store data. For each PVFS file handled by the daemon, a local file is created on an existing local file system. These files are accessed by using standard UNIX `read()`, `write ()`, and `mmap ()` operations. Hence, all data transfer occurs through the kernel block and page caches and is scheduled by the kernel I/O subsystem.

Figure 6.2 shows how the example file `/pvfs/foo` is distributed in PVFS based on the metadata. Note that although there are six I/O nodes in this example, the file is striped across only three I/O nodes, starting from node 2, because the metadata file specifies such a distribution. Each I/O daemon stores its portion of the PVFS file in a file on the local file system on the I/O node. The name of this file is based on the inode number that the PVFS system assigned to the file (in our example, 1092157504).

As mentioned above, when application tasks (clients) open a PVFS file they are returned the locations of the I/O daemons. The clients then establish connections with the I/O daemons directly. These connections are used strictly for data requests. When a client wishes to access file data, the client library sends a description of the file region being accessed to the I/O daemons holding data in the region. The daemons determine what portions of the requested region they have locally and perform the necessary data transfers using TCP/IP.

Figure 6.3 shows an example of how one of these regions, in this case a simple-strided region, might be mapped to the data available on a single I/O node. The intersection of the two regions defines what we call an *I/O stream*. This stream of data is transferred in logical file order across the network connection. By retaining the ordering implicit in the request and allowing the underlying stream protocol to handle packetization, no additional overhead is incurred with control messages at the

application layer.

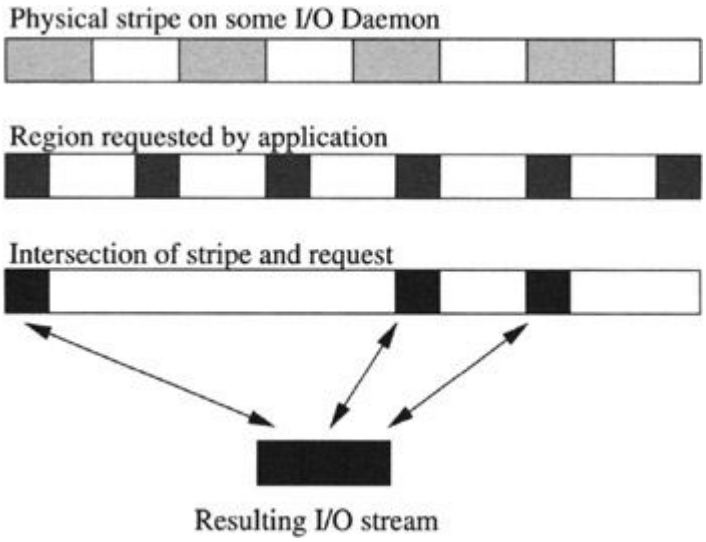


Figure 6.3: I/O stream example

Figure 6.4 shows in greater detail what happens when a client accesses data from PVFS I/O daemons. In gray we see the data that was requested, which corresponds to the region described in Figure 6.3. In black and light gray we see the portions of this data that are stored on the two I/O nodes across which this file is striped.

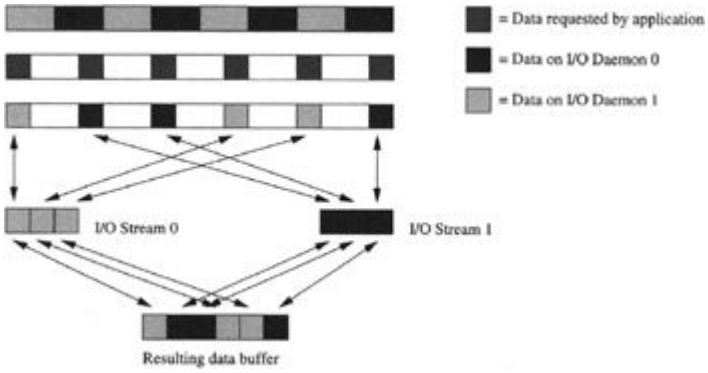


Figure 6.4: File access example

When this region is accessed, the I/O daemons each send back an I/O stream containing the requested data that they possess. These two streams are then merged into the application data buffer on the client.

6.3.3 PVFS request processing

The PVFS request processing mechanism is the component of PVFS that implements the scheduling policy. Here we provide a short overview of the PVFS request processing implementation. First we cover how PVFS receives requests, how these are processed, and the structure in which they are stored for service. Next we discuss how the system handles multiple simultaneous requests. We concentrate on read operations in this discussion, and we detail the actual system calls used by the iods to perform local file system and socket accesses.

6.3.3.1 Receiving and queuing requests

Since all PVFS I/O is currently performed over TCP, all PVFS communication is through the UNIX sockets interface. PVFS I/O servers maintain a set of open sockets that are checked for activity in a loop. One of these sockets is an "accept" socket that is used by clients to establish connections for service. The other two possible states for an open socket are that it is connected but has no outstanding request or that it is in active use for servicing a request.

PVFS I/O servers are single-threaded entities that rely on the `select()` call to identify connections that are ready for service. One of the sockets that the server queries is the accept socket. When this socket (or any other open socket not involved in a request) is ready for reading, the I/O server attempts to receive an I/O *request*. Requests are messages sent by application tasks (clients) asking that some operation be performed on their behalf.

The request is parsed after reception, and if the request requires data transfer, a *job* is created to perform the necessary I/O. Figure 6.5 shows the job data structure as it is being created to service a request.

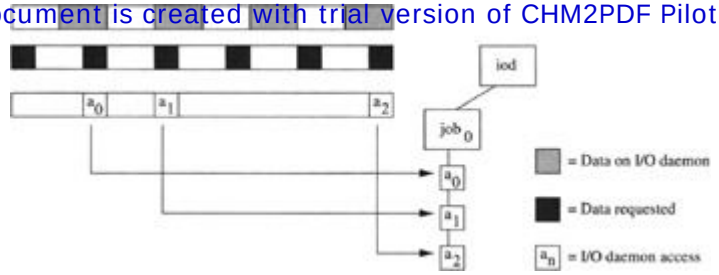


Figure 6.5: Creating accesses from a request

The job is associated with a socket and file, and attached to the job is a list of *accesses*. Accesses are data transfers that must be performed to service the request. A job may have tens or hundreds of accesses.

First the I/O server allocates the job structure and breaks the request into contiguous accesses based on the intersection of the requested data and the data available locally on the server. This is the process shown diagrammatically in Figure 6.5. Following this an acknowledgment is prepended to the access list for passing status information back to the client (i.e., EOF reached). This job is then added to the collection of jobs that the I/O server is processing.

Figure 6.6 shows multiple jobs in service simultaneously. As the I/O server processes these jobs, accesses associated with the job are updated and removed when completed. In this example, Job 1 has already had its acknowledgment sent, so it is no longer present on the access list.

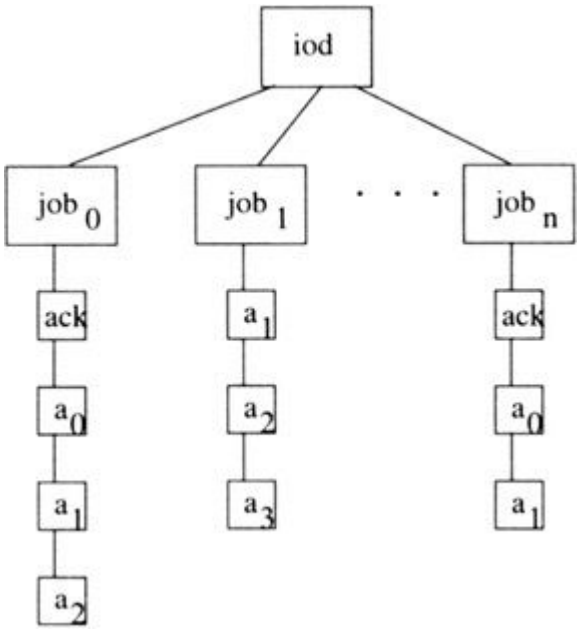


Figure 6.6: Jobs in service

6.3.3.2 Servicing requests

Typically each task in a parallel application will send a request to each I/O server when performing an I/O operation, resulting in a job on each server. It is easy to imagine that for a large parallel application a large number of jobs might be in service on an I/O server at one time.

This large number of jobs, for which there are by definition no interjob dependencies, provide us with an opportunity to optimize by selecting the order in which jobs will be serviced. I/O servers, when not idle, sit in a loop servicing requests:

```
while (job list not empty) {
   select a job to service
   make progress on accesses for selected job
}
```

A job can be selected by any means we wish, including examining characteristics such as size or next access type and position. This gives us the flexibility to implement our scheduling algorithms, which in turn will influence the system by choosing what jobs will be serviced and in what order.

Once a job is selected, the server can perform all or part of the sequence of accesses for the selected job before selecting another job to service. Generally, however, the server should not block waiting for any single job to complete when other jobs could be serviced. Thus, the server normally performs only the accesses or parts of accesses for a selected job that will not cause the server to block.

The I/O server refers to the access list of the selected job in order to determine what operation should be performed next. In the case of a read operation the I/O server first uses `mmap ()` to map the data region into its address space. Mapping is

performed on a region of 128 Kbytes in the implementation tested, which through testing was found to be a reasonable trade-off between mapping too large a region and performing too many mapping operations on this particular system. Once the region is mapped into memory, `send ()` is used to send the data from the desired region to the remote host. The `O_NONBLOCK` flag is set on the socket by using `fcntl ()` prior to sending data so that the server will not block on the socket. When a new region of the file is needed by this connection for I/O, the old region is unmapped with `munmap ()` before the new region is mapped.

6.3.4 Limitations

While PVFS is the most complete open-source parallel file system available and our best option for experimentation, its architecture does place some limitations on our ability to make scheduling decisions and to implement previous schemes.

First and foremost, PVFS servers operate at the user level. This means that all scheduling actions are in some sense indirect; we can put data into a socket buffer or perform a `write ()` call to request that data be stored on disk, but in the end the kernel makes the final decision on when operations happen. This is particularly troublesome in the case of writes; it is very difficult to force the Linux kernel to write data to disk.

Similarly, since iods store data in files, it is more natural to make spatial locality decisions based on file offsets. It is possible that blocks in the file are not placed sequentially, but previous work has shown this technique to be effective [SEA 95]. In our experiments we perform all operations on a single file in order to make the most of file offset information. The use of `mmap ()` for reading data and `write ()` for writing data prevents servers from truly knowing when they will block. They instead rely on the kernel buffering of data to help provide overlap between disk and network I/O.

At the time these tests were performed, PVFS did not support generalized noncontiguous requests. Thus we were unable to use noncontiguous requests for our random access workload. Instead multiple contiguous requests were used, and this approach limits the ability of the server to organize the data movement. Finally, the stream-based data transfer method implemented in PVFS constrains the order in which data may be returned to the client. This places an additional constraint on the server.

These characteristics do inhibit our ability to extract the highest performance from the underlying components. Our goal here, however, is not to show the highest possible performance but instead to show that matching scheduling algorithm to workload can provide a performance win. This list of limitations then serves as a starting point for further improvement of the PVFS system.

6.4 Scheduling algorithms

For our experiments we have implemented four scheduling algorithms that work with PVFS. We designate these algorithms *Opt 1-4*. The first algorithm is the default algorithm for PVFS and is optimized for network access. The other three are increasingly disk oriented, focusing on reducing disk access time. As previously mentioned, PVFS cannot send data out of order for a given job, but it can control the order that multiple jobs are serviced, and these different algorithms reflect this.

Opt 1 is the PVFS default stream-based algorithm. Using this algorithm, the I/O server first checks to see which network connections are ready for service using a `select ()` call and then services each ready socket in FCFS order until the connection is no longer ready for service. When all ready sockets have been serviced, this process is repeated. Because of limited buffering in the network subsystem, this algorithm tends to do a relatively good job of load-balancing service to the various jobs. It does not consider disk access order at all, and may result in significant disk head movement when servicing multiple requests. *Opt 1* has the advantage of being the only algorithm that does not need to sort the jobs, making its processing phase the fastest of the four algorithms.

Opt 2 is a modification of *Opt 1.* As in *Opt 1*, first the server selects all sockets ready for service; then it sorts the sockets based on the offset of the next access from the start of the file. Sorting starts at the last offset accessed for the file (O_{last}), continues to the largest offset for a ready job, and then continues with the jobs whose offsets are smaller than the last offset. In our experiments, all requests are accessing the same file, and if the operating system does a reasonable job of clustering file data on the disk, access to disk should be in a more efficient order than in *Opt 1.*

Opt 3 further modifies *Opt 2* by removing some jobs that are ready for network service because their file offset differs too much from the other jobs. Under this option, a logical window is defined that spans file offsets in a range around the last offset accessed. The center of the window is defined to be the last offset accessed, so for a given window size W_{sz} and last offset O_{last}, values in the range of $O_{last} \pm \frac{W_{sz}}{2}$ are inside the window. The value of W_{sz} is selectable at compile time. With this optimization all requests that are "close" together are serviced, while requests that would access a distant part of the file are not. In the event that no requests fall into this window, the closest request is serviced instead.

Opt 3 is based on a window scan scheduling algorithm (WSCAN) and tends to allow the system to use spatial locality and caching more effectively, especially when the operating system is prefetching based on file offset. On the other hand, this algorithm might not load balance as well as *Opt 1* or *Opt 2* because some jobs that are ready for service might not get serviced for some time. In addition, one has to consider the potential for starvation when a job is specifically excluded from service. Starvation is possible with this algorithm; however, this algorithm will never wait on a job that is not ready as long as a ready job is available.

Opt 4 is the only algorithm that does not consider whether a job is ready for network service. This algorithm sorts all of the jobs based on the offset of the next access, starting from the offset of the last access, and services the jobs in that order, waiting for a network connection to become ready when necessary. This is an implementation of the shortest seek time first (SSTF) algorithm [DEN 67]. This algorithm is extremely likely to starve some jobs, and we will see the results of this when we later discuss fairness. The purpose of including *Opt 4* in our study is to ascertain the maximum possible benefit we can obtain from disk-oriented scheduling within the PVFS system.

Consider the following example. Assume there are five jobs, each accessing the same file. Assume that the last access was at file position 100. In each row, if a number is present, then the socket for that job is ready for service, and the number indicates the file offset being accessed. The schedule shows the jobs that will be serviced, in order. Note that the rounds shown are not intended to represent a fixed amount of time; thus the number of rows does not imply that the resulting schedule necessarily takes longer to complete. Rather, this shows the different scheduling rounds corresponding to passes through the jobs.

Under Opt 1, the ready jobs are scheduled FCFS in each round (Table 6.1). Under *Opt 2*, the ready jobs are scheduled in offset order in each round (Table 6.2). The schedule for *Opt 3* (Table 6.3) bears some explanation. In the first round all three ready jobs fall within the window, so all they are scheduled in offset order. In the second round, Jobs 2 and 3 are within the window, but Job O is not. These jobs are serviced in file offset order. In the third round Jobs 1 and 4 are within the window and are scheduled for service. In the fourth round no jobs are within the window, so the closest (and only) job, Job 0, is serviced. In the fifth round Job 0 is still within the window and service is completed. Finally, under *Opt 4* (Table 6.4), the job with the next offset larger than the previous is scheduled, even if that job isn't ready yet. Thus all blocks are processed in order unless a job arrives after processing has already passed its first offset.

Table 6.1: Example of *Opt 1* scheduling

Round	J_0	J_1	J_2	J_3	J_4	Schedule (in order)
r_0		100	400		300	J_1, J_2, J_4

r_1	900		500	200		J_0, J_2, J_3
r_2	1000	300			400	J_0, J_1, J_4

Table 6.2: Example of *Opt 2* scheduling, starting with O_{last}=100

Round	J_0	J_1	J_2	J_3	J_4	Schedule (in order)
r_0		100	400		300	J_1, J_4, J_2
r_1	900		500	200		J_2, J_3, J_0
r_2	1000	300			400	J_0, J_1, J_4

Table 6.3: Example of *Opt 3* scheduling with W_{sz} =600, starting with O_{last}=100

Round	J_0	J_1	J_2	J_3	J_4	Schedule (in order)
r_0		100	400		300	J_1, J_4, J_2
r_1	900		500	200		J_3, J_2
r_2	900	300			400	J_1, J_4
r_3	900					J_0
r_4	1000					J_0

Table 6.4: Example of *Opt 4* scheduling, starting with O_{last} =100

Round	J_0	J_1	J_2	J_3	J_4	Schedule (in order)
r_0		100	400		300	J_1
r_1	900		400	200	300	J_3
r_2	900	300	400		300	J_1, J_4
r_3	900		400		400	J_2, J_4
r_4	900		500			J_2
r_5	900					J_0
r_6	1000					J_0

6.5 Workloads

In Section 6.3 we discussed PVFS, the parallel file system used in our experiments. In this section we discuss the workloads we examined in order to ascertain the effectiveness of our scheduling algorithms at servicing three widely different test patterns:

- single block accesses
- strided accesses
- random block accesses

These access patterns are seen in some traditional applications, particularly ones operating on dense multidimensional matrices. Figure 6.7 shows the general pattern of access for these workloads shown as row-major two-dimensional structures.

Figure 6.7: Test workloads

In all cases an MPI application was used to create a set of application tasks that independently access a PVFS file system using the native PVFS libraries. In all tests a single PVFS file was used to store data.

In the single block access tests, contiguous regions of the data file were simultaneously accessed by each task. The tasks synchronize before the access, and the times to complete access were recorded. We consider the longest service time of any one task to be the application service time, as this is the time the application as a whole would have to wait if operations were collective. We also calculate the mean service time for all tasks and the variance of task service time. Patterns such as these are seen in applications accessing dense matrices in a block manner, in checkpoint applications, and in some out-of-core applications.

In the strided access tests, multiple noncontiguous regions of the data file were simultaneously accessed by each task using a single, simple-strided operation. Again, the tasks synchronize before the access, and the times to complete the operation were recorded. As before, we consider the longest task service time to be the application service time, and we also calculate mean service time and the variance. Strided accesses are often seen as a result of row cyclic distribution of data sets and access to portions of records of a fixed size.

The purpose of the random block access tests is to observe the system serving an application with an irregular access pattern. The file is logically divided into a number of blocks, and these blocks are randomly assigned to the tasks in the application such that each task will access an equal number of blocks. Tasks are synchronized before any accesses begins, and tasks access all blocks in random order using a native PVFS operation to access each block, one block at a time. The time to access all blocks is recorded for each task, and the largest of these total times is considered the application service time. Mean service time and variance are also calculated. This pattern might be created by an application accessing pieces of a multidimensional data set or reading arbitrary records from a large database.

In all cases, our goal is to analyze the effects of the four scheduling algorithms on the performance of the system, using the three metrics application service time, mean task service time, and task service time variance. All of these metrics are important in one situation or another. For a system serving single parallel applications, application service time might be the most appropriate metric. For a system running multiple parallel applications or many serial ones, mean task service time might be a more appropriate metric. Task service time variance is the variance observed between the service times of tasks concurrently accessing the system. This value is an indicator of fairness; a low value indicates that service is distributed fairly between jobs, while a high variance often indicates that starvation is occurring. This is of particular importance in real-time applications.

6.6 Experimental results

The Beowulf machine on which this work was performed is a 17-node cluster at Clemson University. The cluster is configured as follows. Each node has a single Pentium 150 MHz CPU, 64 Mbytes EDO DRAM, 64 Mbytes local swap space, a 2.1 Gbyte IDE disk, and Tulip-based 100 Mbit Fast Ethernet card. The nodes are connected by an Intel Fast Ethernet switch in full-duplex mode.

One node runs the PVFS manager daemon and handles interactive connections while the other nodes are used as compute nodes, I/O nodes, or both. Each node runs Linux v2.2.13 with tulip driver v0.89H. The IDE disks provide approximately 4.5 Mbytes/sec with sustained writes and 4.2 Mbytes/sec with sustained reads, as reported by Bonnie, a popular UNIX file system performance benchmark [BRA]. When idle, approximately 8 Mbytes of memory are used on each node by the kernel and various system processes, including PVFS, leaving approximately 56 Mbytes of space that could be used by the system for I/O buffers. The window size for tests with *Opt 3* was set to 56 Mbytes. Our test applications were compiled with MPICH v1.2.0.

For these tests 2 nodes were used as I/O nodes, and 14 nodes were used for computation. This combination allowed us to separate the I/O nodes from the compute ones, to provide a number of simultaneous jobs that the I/O nodes can schedule, and to ensure that no single-disk optimizations are used on the I/O nodes (mapping PVFS file locations to local file ones is simpler in the single-disk case).

Varying scheduling algorithms for write workloads showed only limited benefit, so this data is not presented here. Interested readers are directed to [ROS 00] for this data. For read tests, before each run a local data file larger than the size of a node's memory was read in its entirety on each I/O node to remove all PVFS file data from cache. A separate run of read tests was additionally performed in which we allowed file data to remain in cache (i.e., did not read a local data file between runs). These results are compared with small accesses without cache in the following subsections as well.

We first discuss the application and task service time metric for each of the tests, presenting output for all four of the algorithms described earlier in this work. The total data accessed on a single I/O node is shown on the x axis, and service time is provided on the y axis. The data presented is the average of three test runs. Following this discussion we cover the issue of fairness with respect to our algorithms and the tested workloads.

6.6.1 Single-block-accesses

The results for single block accesses are shown in Figure 6.8. When data is uncached we see that *Opt 4* provides by far the lowest average task service time, beating the worst performers by as much as 28%. Recall that *Opt 4* is our algorithm most similar to disk-directed I/O; only the request closest to the last accessed file position will be serviced. Clearly in this case disk resources are used more effectively with *Opt 4.* However, application service times are consistent across all algorithms.

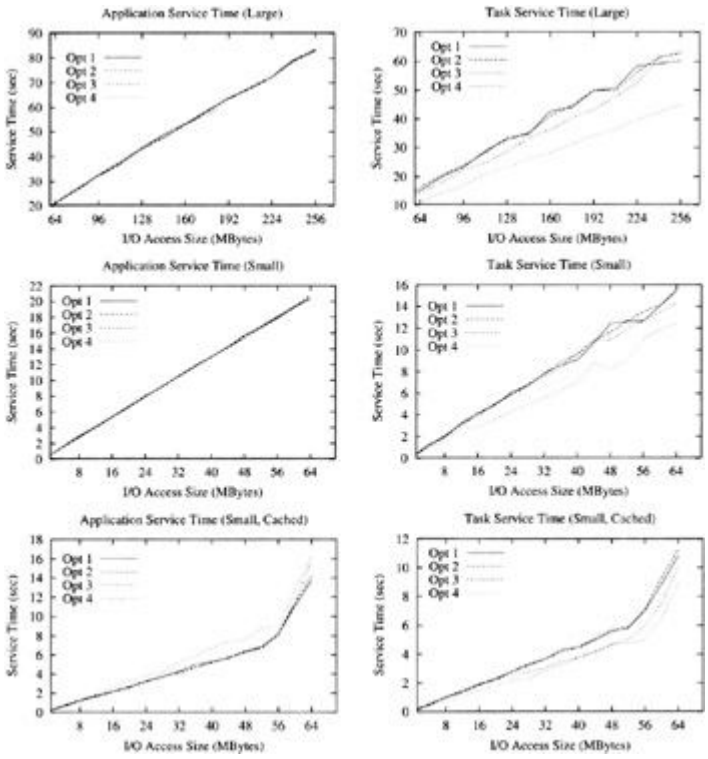


Figure 6.8: Single block read performance

When file data is cached, we see that *Opt 4* still results in the best task service time (beating the worst performer by 30%); however, this comes at the cost of a substantially higher application service time than the other algorithms. In this case *Opt 3* seems to be a more appropriate overall choice. Recall that *Opt 3* relaxes the strict ordering of requests, allowing for jobs within a window to be serviced and always allowing the nearest ready job to be serviced. When cached data is available, *Opt 3* provides a better application service time while also resulting in a competitive mean task service time.

When examining the small, cached service time graphs, one can see a uniform change in performance at approximately 56 Mbytes. This is the point at which we begin to exceed our cache size and start hitting disk. This trend will be seen throughout all the test results.

6.6.2 Strided accesses

Results for strided accesses are shown in Figure 6.9. For our strided access pattern we arbitrarily chose to access 16 disjoint regions with each task access. The size of the disjoint regions was varied throughout the tests.

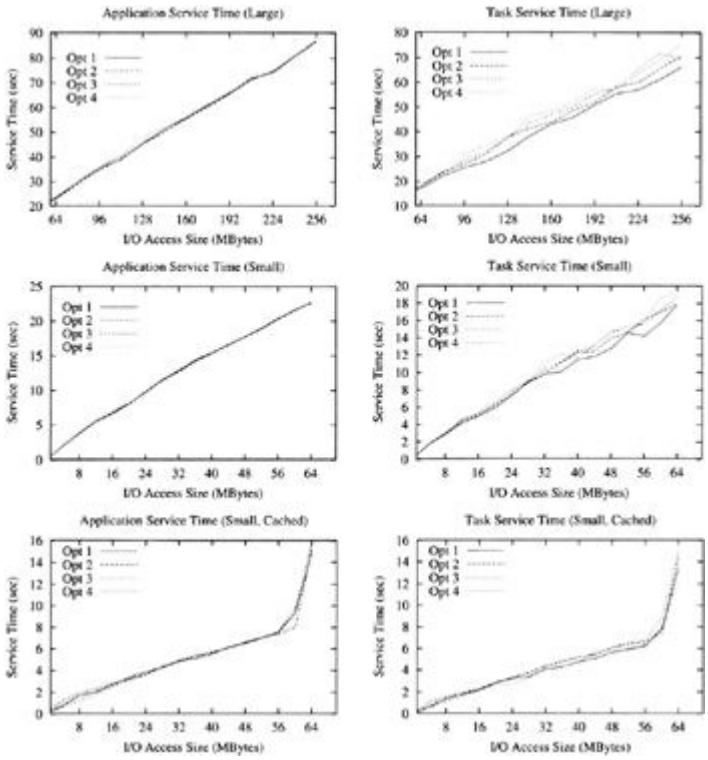


Figure 6.9: Strided read performance

When servicing uncached strided read requests, we see that *Opt 1* provides the lowest mean task service time by as much as 14%, most likely because file data is interleaved between the application tasks, resulting in *Opt 1* performing similarly to *Opt 4*, but without its ordering restrictions. When data does reside in cache, we see less benefit from using *Opt 1* over other algorithms, although it does appear to still be the best choice. All algorithms result in approximately the same application read service times over the wide range of access sizes.

The file byte ordering limitation imposed by PVFS is particularly inhibiting for disk-oriented algorithms servicing this type of workload. Since in practice all jobs are not started simultaneously, it is likely that the first job to arrive will be partially serviced before others are started. These new jobs might have data located near the data accessed for the first job, but those portions of the new jobs may not be serviced until their point in the byte ordering is reached.

6.6.3 Random block accesses

We chose to study random block access in addition to focusing on known patterns. An interesting characteristic of these tests is that only a fraction of the total data to be accessed is being requested by jobs in service at any one time because multiple operations are required to access the randomly distributed blocks (using native PVFS calls). This is in contrast to the previous tests, where all data to be accessed is requested in single calls. The result is that the total size of requests at any point in time is no more than S_{tot}/N_{blks}, where S_{tot} is the total amount of data that will be accessed and N_{blks} is the number of blocks into which the data is split (per task). Tests were run for N_{blks} values of 16 and 32. Results for both were similar, so only the results from 32 blocks per task are presented here. Interested readers are directed to [ROS 00] for this data.

Figure 6.10 presents the results for 32 blocks per task. This is the first set of tests for which we see a significant difference in uncached application service times between algorithms. This is a clear indicator that we are reducing the amount of work performed by the underlying I/O system using the disk-oriented algorithms *Opt 3* and *Opt 4*. These two algorithms also outperform the others in task service time for large accesses. When caching is in effect, we again note that *Opt 1* becomes extremely competitive, outperforming *Opt 4* by as much as 10%.

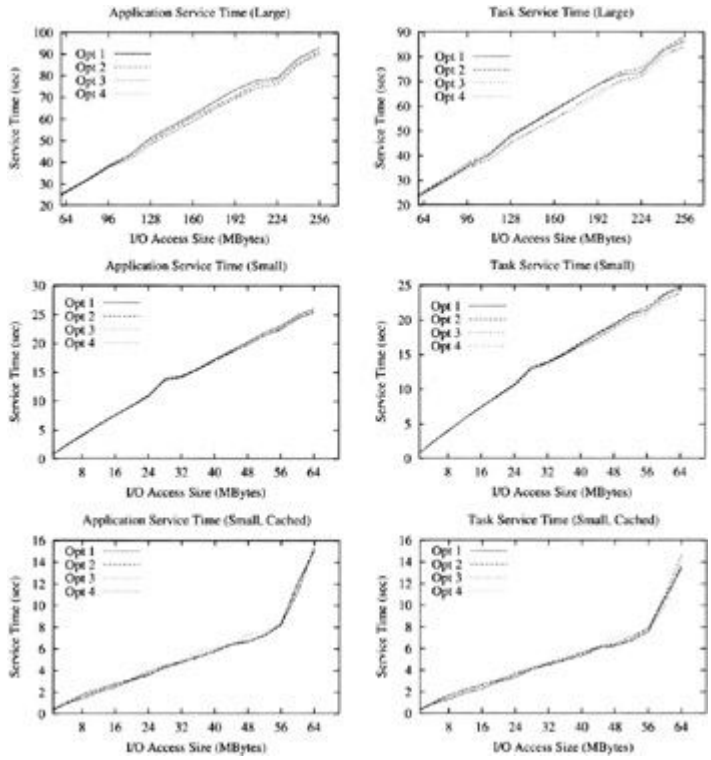


Figure 6.10: Random (32 block) read performance

6.6.4 Fairness

For some applications it is highly desirable for service times to be predictable. For these applications fairness is of extreme importance, as starvation will lead to unpredictable service times. In Figure 6.11 we show the task service time variance for our tests. We show the results for small, cached data and large accesses only; small uncached results followed the trends of the large accesses.

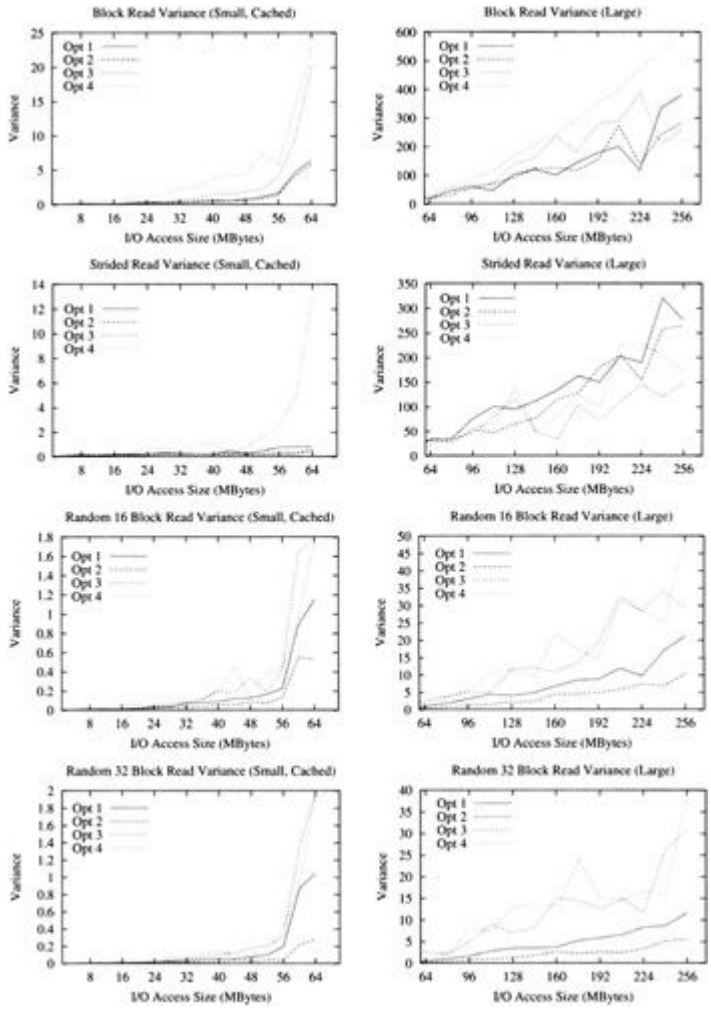


Figure 6.11: Task service time variance for reads

When looking at the small, cached graphs, we observe a uniform trend of spikes in the variance graphs as we enter the 56-64 Mbyte range. This is to be expected because it is at this point that accesses first begin to result in cache misses.

Additionally we see that in general *Opt 1* and *Opt 2* provide better (lower) variance than do *Opt 3* and *Opt 4*. We expected this because *Opt 1* and *Opt 2* both service all ready jobs on each pass, while *Opt 3* and *Opt 4* allow certain jobs to starve. *Opt 2* also tends to provide more predictable performance than *Opt 1*; this is undoubtedly due to its more fair method of cycling through file locations.

The notable exception to this is large, strided access. In this case *Opt 3* and *Opt 4* provide the most predictable performance. This is due to their more strict enforcement of ordering; by ordering accesses by their file locations, they enforce fair service when job data happens to be organized in a strided manner.

As noted previously, *Opt 3* and *Opt 4* both introduce the possibility of starvation. Particularly in the case of *Opt 4* workarounds to avoid this situation would need to be added if the algorithm were to be used in a production system. The *Opt 3* algorithm already implements a degree of starvation-avoidance, and based on this we believe that the addition of such workarounds would have a minimal impact of performance in common workloads.

6.7 Conclusions and future work

As a whole we see that we are able to affect application service time in only a small number of cases, and in general our scheduling changes has little effect on write workloads. This is not completely surprising considering the limitations discussed in Section 6.3.4.

However, we consistently see benefits to applying certain algorithms in read cases with respect to task service time. In particular, for situations where uncached contiguous regions are being serviced, *Opt 3* and *Opt 4* show the best performance.

On the other hand, for cases where significant fractions of data are cached, we see that *Opt 1* performs the best. This is likely to be in part due to the algorithm itself and in part due to the time it saves by not sorting jobs in service.

For our strided read workload we see that *Opt 1* performs best as well. This is partially due to the implicit interleaving in the requests. However, it is likely that our predefined ordering of request data is also a factor. By this we mean that request data is always returned to the requesting task by PVFS in order of monotonically increasing file byte offset. This constrains the order in which we can service the pieces that make up strided requests, which in turn limits the ability of more disk-oriented algorithms to best access the disk.

If one is most concerned with "fair" service, *Opt 2* is the best choice. It provided the lowest variance for almost all the tested cases, with little loss in application or task service time. The only exception to this is the strided case with large accesses, in which *Opt 3* is the best choice.

Overall we see that no single algorithm performs best over the range of workloads, but instead that algorithms tend to be appropriate to a workloads fitting certain characteristics. This result indicates that rather than relying on a single algorithm, servers should instead have a collection of algorithms at their disposal. These algorithms could be selected by administrators based on expected use, but a more effective approach would be to automatically select algorithms in response to workload characteristics at run time.

Adaptive selection of policies for caching and prefetching have already been developed [MAD 96, MAD 97]. Our intention is to build a complementary system for selecting scheduling algorithms. A behavioral model incorporating workload characteristics such as the extent and size of requests in service and system effects such as available cache would be coupled with heuristics for algorithm selection based on this work. One concern with such a solution is that the overhead of performing the calculations at runtime might outweigh the potential gains, but previous work in kernel-level scheduling on similar machines indicates that performing additional calculations at runtime should be feasible [GEI 97].

We have recently implemented noncontiguous requests for PVFS. This capability extends the application's ability to describe the overall desired I/O pattern to the server. which should enable us to better schedule service. Additional studies are necessary to validate this claim.

Acknowledgements

This work was funded in part by National Aeronautics and Space Administration grants NAG5-3835 and NGT5-21 and in part by the Mathematical. Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

6.8 Bibliography

[BOR 93] Bordawekar R., Del Rosario J. M., Choudhary A., *"Design and Evaluation of Primitives for Parallel I/O"*, *Proceedings of Supercomputing '93*, Portland, OR, IEEE Computer Society Press, p. 452-461, 1993.

[BRA] Bray T., *"Bonnie File System Benchmark"*, Available online at the following URL: http://www.textuality.com/bonnie/.

[CAR 00] Carns P. H., Ligon III W. B., Ross R. B., Thakur R., *"PVFS: A Parallel File System for Linux Clusters"*, *Proceedings of the 4th Annual Linux Showcase and Conference*, Atlanta, GA, USENIX Association, p. 317-327, October 2000.

[CHO 94] Choudhary A., Bordawekar R., Harry M., Krishnaiyer R., Ponnusamy R., Singh T., Thakur R., PASSION: Parallel And Scalable Software for Input-Output, Report num. SCCS-636, ECE Dept., NPAC and CASE Center, Syracuse University, September 1994.

[DEN 67] Denning P. J., *"Effects of Scheduling on File Memory Operations"*, *AFIPS Spring Joint Computer Conference*, April 1967.

[GEI 97] Geist R., Ross R., *"Disk Scheduling Revisited: Can O(N^2) Algorithms Compete?"*, *Proceedings of the 35th Annual ACM Southeast Conference*, April 1997.

[KOT 95] Kotz D., Nieuwejaar N., *"File-System Workload on a Scientific Multiprocessor"*, *IEEE Parallel and Distributed Technology*, vol. 3, num. 1, p. 51-60, IEEE Computer Society Press, Spring 1995.

[KOT 97] Kotz D., *"Disk-directed I/O for MIMD Multiprocessors"*, *ACM Transactions on Computer Systems*, vol. 15, num. 1, p. 41-74, ACM Press, February 1997.

[KRY 93] Krystynak J., Nitzberg B., *"Performance Characteristics of the iPSC/860 and CM-2 I/O Systems"*, *Proceedings of the Seventh International Parallel Processing Symposium*, Newport Beach, CA, IEEE Computer Society Press, p. 837-841, 1993.

[LIG 96] Ligon W. B., Ross R. B., *"Implementation and Performance of a Parallel File System for High Performance Distributed Applications"*, *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, IEEE Computer Society Press, p. 471-480, August 1996.

[MAD 96] Madhyastha T. M., Reed D. A., *"Intelligent, Adaptive File System Policy Selection"*, *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, IEEE Computer Society Press, p. 172-179, October 1996.

[MAD 97] Madhyastha T. M., Reed D. A., *"Input/Output Access Pattern Classification Using Hidden Markov Models"*, *Proceedings of the Fifth Workshop on Input/Output in Parallel and Distributed Systems*, San Jose, CA, ACM Press, p. 57-67, November 1997.

[NIT 92] Nitzberg B., Performance of the iPSC/860 Concurrent File System, Report num. RND-92-020, NAS Systems Division, NASA Ames, December 1992.

[RID 97] Ridge D., Becker D., Merkey P., Sterling T., *"Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs"*, *Proceedings of the 1997 IEEE Aerospace Conference*, 1997.

[ROM] *"ROMIO: A High-Performance, Portable MPI-IO Implementation"*, http://www.mcs.anl.gov/romio.

[ROS 00] Ross R. B., Reactive Scheduling for Parallel I/O Systems, PhD thesis, Dept. of Electrical and Computer Engineering, Clemson University, Clemson, SC, December 2000.

[SEA 95] Seamons K. E., Chen Y., Jones P., Jozwiak J., Winslett M., *"Server-Directed Collective I/O in Panda"*, *Proceedings of Supercomputing '95*, San Diego, CA, IEEE Computer Society Press, December 1995.

[TAK 99] Taki H., Utard G., *"MPI-IO on a Parallel File System for Cluster of Workstations"*, *Proceedings of the First IEEE International Workshop on Cluster Computing*, IEEE Computer Society Press, p. 150-157, 1999.

Chapter 7: Design and Implementation of a Large Virtual NVRAM Cache for Software RAID

Xubin He, Department of Electrical and Computer Engineering, Tennessee Technological University, Cookeville, TN 38501,
USA
Qing Yang, Department of Electrical and Computer Engineering, University of Rhode Island, Kingston, RI 02881, USA

7.1 Introduction

In order to overcome the small write problem in RAID5, especially software RAID5, we have designed and implemented a software RAID with a large virtual NVRAM cache under the Linux kernel. Because no additional hardware is needed to implement our write cache, we named it Virtual NVRAM Cache or VCRAID for short. The main idea is to use a combination of a small portion of the system RAM and a log disk to form a hierarchical cache. The log disk can be either a dedicated physical disk or several partitions of disks in the RAID. Since the log disk quickly absorbs write data from the small RAM, this hierarchical cache appears to the host as a large nonvolatile RAM. A prototype VCRAID implemented under the Linux kernel has been tested for an extended period of time to show that it functions properly. Performance measurements have been carried out using typical programs and popular benchmarks such as ServerBench 4.1, PostMark and Bonnie. Our measurements show that the VCRAID has superb performance advantages over the built in software RAID shipped with the Linux package. Depending on the workload characteristics, performance gains due to VCRAID range from 67.7% to an order of magnitude. For applications that have data locality, we observed up to a factor of 16 performance improvements in terms of user response time. In many situations, VCRAID achieves similar performance as RAID0 and some time better than RAID0 indicating that VCRAID realizes the maximum potential to hide small write problems in RAID5.

Software RAID has become very popular for applications that require reliable and economic storage solutions as Ecommerce emerges. Most commodity operating systems such as Windows, Solaris, and Linux have built in software RAIDs. While these built in software RAIDs provide immediate performance benefits at lowest cost, most software RAID installations shy away from RAID5 configuration, the most popular RAID configuration in storage industry, and therefore lose the benefit of high reliability of RAID. The main reason why RAID5 is not installed in software RAID is the performance penalty resulting from small write operations. Each such a small write requires 4 disk operations: reading old data, reading old parity, writing new data, and writing new parity. As a result, for workloads consisting of a mix of read and write operations, software RAID5 shows very poor performance and becomes impractical.

The most popular and practical solution to small write problems is to use large write cache. Modern RAID systems make extensive use of nonvolatile RAM (NVRAM) write caches to allow fast write [CHE 94][HOU 97][MEN 93][TRE 95], or asynchronous write. Write requests are acknowledged before they go to disk. Such write caches significantly reduce user response times of RAID. Large write caches can also improve system throughput by taking advantage of both temporal locality and spatial locality of general workloads. Treiber and Menon reported that write caches could reduce disk utilization for write by an order of magnitude when compared to standard RAID5 [TRE 95]. In fact, some researchers have argued that the use of large caches in RAID systems has rendered debates over the best RAID level irrelevant [COO 96].

While most commercial hardware RAID systems have large write caches for better performance, there is no readily applicable cache solution for software RAID or doityourself RAID [ASA 95]. This is because that write cache has to be nonvolatile to be reliable. Adding a nonvolatile RAM into software RAID not only is costly but also requires special hardware devices to interface to the disks. Our objective here is to present a simple software cache solution that can be embedded into the software RAID package shipped with a commodity OS with no additional hardware.

Our approach is to use a combination of a small RAM and a log disk to form a large and nonvolatile write cache. The small RAM (a few megabytes) is obtained from a raw partition of the system RAM. The log disk can reside on one physical disk or on partitions of several disks in the RAID. The combination of the RAM and the log disk form a hierarchical write cache. Disk writes are first collected in the small RAM to form a log that is quickly moved to log disk sequentially so that the RAM is emptied for further writes. It appears to the host that there is a very large RAM cache since there is always space for new writes. At the same time, such a large "RAM" has a very high reliability because the small RAM is used only for collecting log and data stay in the RAM for a very short period of time before moving to cache disk that is nonvolatile and more reliable. This large virtual RAM cache hides the small write penalty of RAID5 by buffering small writes and destaging data back to RAID with parity computation when disk activity is low (We are using a thread to monitor the system kernel activity dynamically similar to the command "top" in Linux. This approach is different from the writeahead logging [GRA 93][MOH 95] in that we use a cache disk which is nonvolatile and inexpensive, and also we do not need to build a new file system as in [CHU 92] [MAT 97][SEL 93][SHI 95].

We have designed and implemented this virtual RAM cache under the Linux operating system. Extensive testing of the implementation has been carried out to show it functions properly. We have also carried out performance measurements and

comparisons with existing software RAID using typical disk I/O bound programs and popular benchmarks such as ServerBench, PostMark and Bonnie. Our measurements on ServerBench show that the RAID5 with our virtual RAM cache has superb performance advantages over the built in software RAID shipped with the Linux package. Up to a factor of 3 performance improvements have been observed in terms of transactions per second. Measurements on Bonnie also show similar performance improvements. When we measure the performance of realistic application programs under Linux, we observed much greater performance gains because realistic applications show data locality. An order of magnitude performance improvement was observed in terms of user response time. In many situations, VCRAID achieves similar performance as RAID0 and some time better than RAID0 indicating that VCRAID realizes the maximum potential to hide small write problems in RAID5.

The paper is organized as follows. In the next section, we present detailed concept and design of the large virtual NVRAM cache. Section 7.3 presents the implementation details. Benchmark and performance results are presented in Section 7.4. We discuss previous related research work in Section 7.5 and conclude our paper in Section 7.6.

PREV < Day Day Up > NEXT

7.2 Architecture and design of the virtual NVRAM cache

In the absence of physical NVRAM cache in software Doityourself RAID, we try to make use of existing system resources and implement a virtual NVRAM cache by means of software. Our idea is very simple. We use a small raw partition of the system RAM and partitions of disks in the RAID to form a cache hierarchy. A thin layer of driver program is inserted between the Linux kernel and the physical device driver for disks. This driver program manages the operation of the cache hierarchy in the way described below.

The general structure of the virtual NVRAM cache has a two level hierarchy: a small RAM buffer on top of a disk called *cache disk*. Small write requests are first collected in the small RAM buffer. When the RAM buffer becomes full or a programmable timer is out, all the data blocks in the RAM buffer are written to the cache disk in several *large data transfers.* Each large transfer finishes quickly since it requires only one seek instead of tens of seeks. As a result, the RAM buffer is very quickly made available again to accept new incoming requests. The twolevel cache appears to the host as a large virtual NVRAM cache with a size close to the size of the cache disk. When the system I/O activity is low, it performs *destaging* operations which computes parity and transfer data from the cache disk to the disk array, referred to as *data disks* or *data disk array.* The destaging overhead is quite low, because most data in the cache disk are shortlived and are quickly overwritten therefore requiring no destaging at all [HU 96]. Moreover, many systems, such as those used in office/engineering environments, have shown significant temporal and spatial data locality giving rise to sufficient long idle periods between bursts of requests. Destaging can be performed in the idle periods therefore will not interfere with normal user operations at all. Since the cache disk is a disk with a capacity much larger than a normal NVRAM cache, it can achieve very high performance with much less cost. It is also nonvolatile thus highly reliable.

The cache disk in the virtual NVRAM cache can be a separate physical disk dedicated for caching purpose. Given the low cost of today's hard drives, it is quite feasible to have one of the disks in the disk array to carry out the cache disk function. The cache disk can also be implemented using a collection of logical disk partitions from disks in the RAID with no dedicated cache disk. Depending on how the cache disk is formed, we have four alternative architecture configurations for *VCRAID* as illustrated in Figure 7.1.

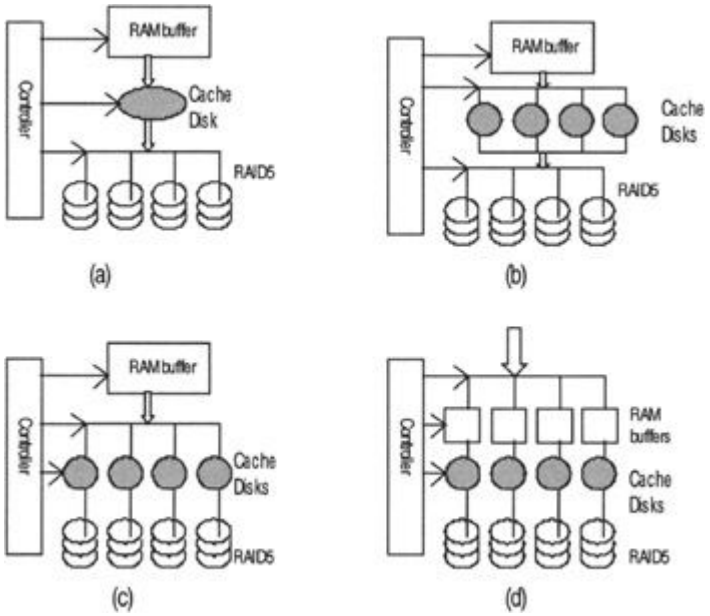


Figure 7.1: Possible approaches to VCRAID. (a) one RAM buffer and one cache disk (b) one RAM buffer and several cache disks (c) one RAM buffer and several cache disks, each cache disk is associated with a disk in the array (d) Several RAM buffers and cache disks, each RAM buffer and cache disk are associated with a disk in the array

In the first configuration, a RAM buffer and a physical cache disk are used to cache data to be written to the entire disk array (Figure 7.1a). This configuration is referred to as *Single Cache Disk (scd)* configuration. All data to be written to the array are first written to the RAM buffer. When the RAM buffer is full or the cache disk is idle, the data in the RAM buffer are transferred to the cache disk by a kernel thread, called *RAIDDestage* thread. The *RAIDDestage* thread combines small write requests writes into large one and write to the cache disk at a time. The data in the cache disk are destaged to the data disk array during the system idle time and/or when the cache disk is full. We can choose the size of the cache disk to be large enough to ensure that most of the destages occur during the system idle time.

The above configuration *(scd)* is effective for a small array consisting of a few disks. With the increase of the number of disks in the array, a single cache disk in the above configuration may become a new system bottleneck since all write operations and some read operations are performed at this cache disk. To avoid this bottleneck problem, the second configuration uses *Multiple Cache Disks (mcd)* as shown in Figure 7.1b. Several cache disks collectively form the cache disk in the virtual

NVRAM hierarchy. These cache disks are logical partitions physically residing on the data disk array. All these logical partitions form a uniform disk space to cache all write data to the data disk array. When the RAM buffer is full or there are idle cache disks, the destage thread is invoked to move data from RAM buffer to one of the cache disks. A roundrobin algorithm is used to determine which cache disk is used next in a log write. Writing to and reading from cache disks can therefore be done in parallel reducing the bottleneck problem.

The above two configurations (*scd* and *mcd*) both have a global cache that cache data for the entire disk array. It is also possible to have private write caches for each disk in the disk array. That is, there is a virtual NVRAM cache for each individual disk in the disk array. Figure 7.1c shows this configuration where a cache disk is associated with each data disk in the array and only cache data for the particular data disk below it. If the RAM buffer is also managed individually for each data disk, then we have the fourth configuration as shown in Figure 7.1d.

PREV < Day Day Up > NEXT

7.3 Implementation

This section presents data structures and algorithms used in implementing the *VCRAID.* We have implemented the *VCRAID* as a loadable module under Linux kernel 2.0.36 based on the *MD* (MultiDevice) driver. The *MultiDevice* kernel module, included as a standard part of the v2.0.x kernels, provides RAID0 (disk striping) and multidisk linearappend spanning support in the Linux kernel. The RAID 1, 4 and 5 kernel modules are a standard part of the latest 2.1.x kernels; patches are available for the 2.0.x kernels and the earlier 2.1.x kernels.

Let us first look at how disk requests are processed on Linux. Take a synchronous write as an example. As shown in Figure 7.2a, a user program issues a write request by calling *fwrite().* The system switches to the kernel mode through the system call *write().* Then the Linux generic block driver accepts the requests through *ll_rw_block()* and issues a request to the lowlevel physical disks through *make_request().*

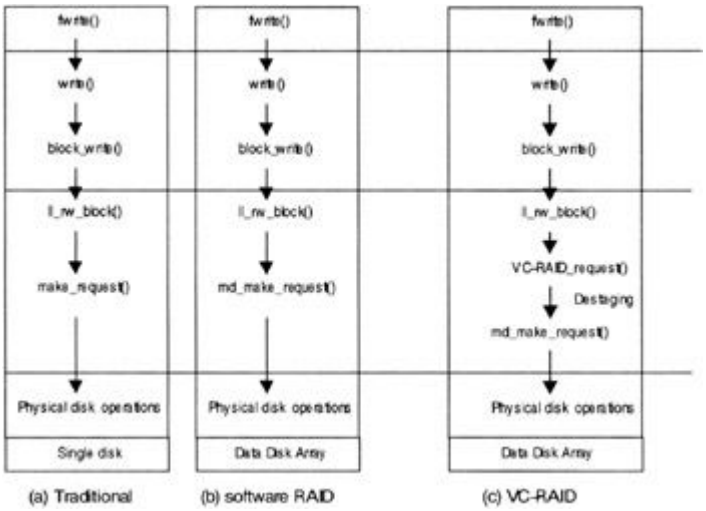


Figure 7.2: Procession of write requests

The physical disk driver finishes the request by *hd_request()* (IDE disks) or *sd_rquest()* (SCSI disks). Figure 7.2b shows the case of software RAID, where *md* driver accepts the requests and determines which disk in the array services the request and computes the parity, finally writes the data and parity to the corresponding disks through low level physical disk drivers.

Figure 7.2c shows how *VCRAID* processes a write request. *VCRAID_request* accepts all requests to the array, and combines small writes into large ones in the RAM buffer firstly. Whenever the cache disk is idle, the data in the RAM buffer are moved to the cache disk in a large write. During the system idle time, data in the cache disk will be destaged to disk arrays.

7.3.1 In memory data structure

During the system initialization, a fixed amount of physical RAM is reserved for our *VCRAID*, all the in memory data structures reside in this RAM area. The in memory data structure includes a Hash table which is used to locate data in the cache, a data buffer which contains several data slots and a number of in memory headers (Figure 7.3). The in memory headers form two lists: the Free List which is used to keep track of free slots and the LRU list which traces the Least Recently Used slots.

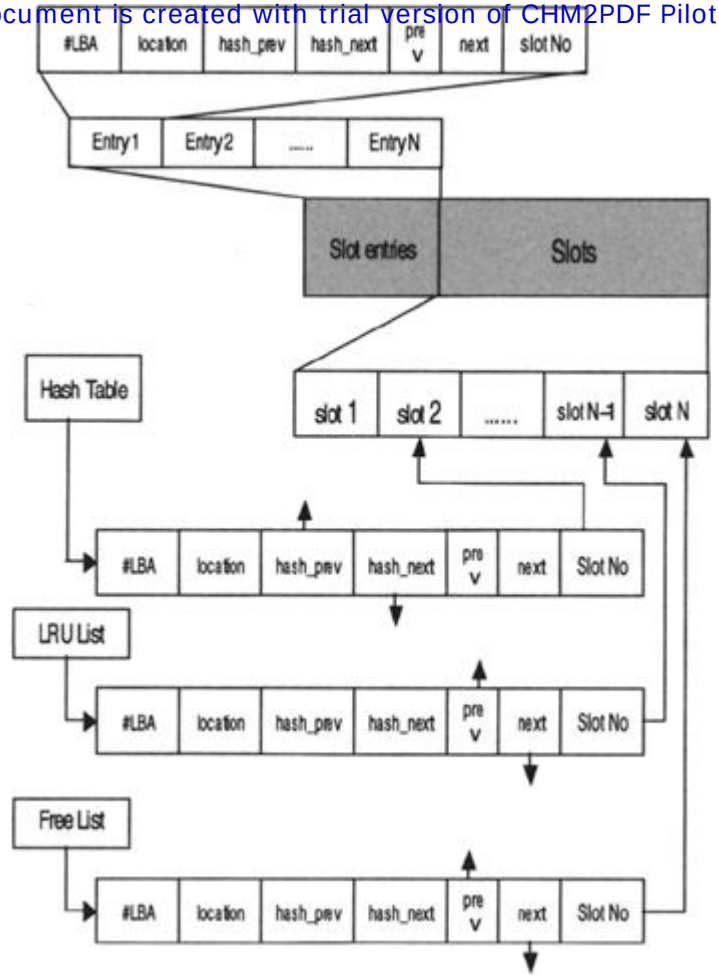


Figure 7.3: RAM buffer layout. RAM buffer consists of slot entries and slots. The hash table, LRU list and Free list are used to organize the slot entries

In our implementation, a data block may exist in one of the following places: the RAM buffer, the cache disk or the data disks. A data Lookup Table is used to keep track of the data location in these places. Since the driver must search the table for every incoming request, it is imperative to make the searching algorithm efficient. In Linux kernel 2.0.36, the default file system is *ext2* file system. When a file system is created by *mke2fs*, the block size, *b_size*, will be fixed. The default is 1024 byte per block, or it can be specified to be 1024, 2048 or 4096. After a file system is created, the LBAs (Logical Block Address) of all requests are aligned to the *b_size* boundary. And all atomic request size equals to *b_size*. We use a hash table to implement the data lookup table with the LBA of incoming requests being the search key, and the slot size being the size of a block, as shown in Figure 7.3. A slot entry consists of the following fields:

- An *LBA* entry that is the LBA of the cache line and serves as the search key of hash table;
- A *location* field is divided into three parts:
    1. A cache disk index (8 bits), used to identify a cache disk in multiple cache disk configuration. It can support up to 256 cache disks. In single cache disk configuration, this field is set to zero;
    2. A state tag (4 bits), used to specify where the slot data is: IN_RAM_BUFFER, IN_CACHE_DISK, IN_DATA_DISK or SLOT_FREE;
    3. A cache disk block index (20 bits), used to specify the cache disk block number if the state tag indicates IN_CACHE_DISK. The size of each cache disk can be up to 1048576 blocks.
- Two pointers (*hash_prev* and *hash_next*) are used to link the hash table;
- Two pointers (*prev* and *next*) are used to link the LRU list and FREE list;
- A *SlotNo* is used to describe the in memory location of the cached data.

7.3.2 Cache disk organization

The organization of cache disk is much simpler compared to that of RAM buffer. The cache disk consists of cache disk headers and an amount of physically consecutive blocks. The block size equals to the slot size in RAM buffer. A cache disk header comprises a number of cachediskblockindex/arrayLBA pairs which are used to track the locations of cached data blocks in the data disk array. The cache disk headers are only for crash recovery purpose and are never accessed during normal operations.

7.3.3 Basic operations

7.3.3.1 Write

After receiving a write request, the *VCRAID* driver first searches the Hash Table. If an entry is found, the entry is overwritten by the incoming write. Otherwise, a free slot entry is allocated from the Free List, and the data are then copied into the corresponding slot, and its address is recorded in the Hash table. The LRU list and Free List are updated. The driver then signals the kernel that the request is complete, even though the data has not been written into the disk array. This immediate report scheme does not cause any reliability problem as will be discussed in section 7.3.4. If user applications set "*O_SYNC*" flag in the write request, the driver writes the data to the cache disk directly and wait until the requests finishes before signalling the kernel that the request is complete.

7.3.3.2 Read

After receiving a read request, the *VCRAID* driver searches the Hash Table to determine the location of the data. In our case, data requested may be in one of three different places: the *VCRAID* RAM buffer, the cache disk(s), or the data disk array. If the data is found in the RAM buffer, the data are copied from the ram buffer to the requesting buffer, and then the driver signals the kernel that the request is complete. If the data is found in the cache disk(s), the data are read from the cache disk into the requesting buffer; otherwise, the *VCRAID* driver forwards the read request to the disk array.

7.3.3.3 Destages

There are two levels of destages: destaging data from the RAM buffer to the cache disk *(Level 1 destage)* and destaging data from cache disk to data disk array *(Level 2 destage)*. We implement a separate kernel thread *RAIDDestage* to perform the destaging tasks. The *RAIDDestage* thread is registered during system initialization and monitors the *VCRAID* states. The thread keeps sleep at most of the time, and is activated when the *VCRAID* driver detects an idle period or the *VCRAID* RAM buffer and/or the cache disk becomes full. *Level 1 destage* has higher priority than the *Level 2 destage*. Once the *Level 1 destage* starts, it continues until the RAM buffer becomes empty and it is uninterruptible. Once the *Level 2 destage* starts, it continues until the cache disk becomes empty, or until a new request comes in. In the later case, the destage thread is suspended, until the driver detects another idle period.

As for *Level 1 destage*, the data in the RAM buffer are written to the cache disk sequentially in large size (up to 64K). The cache disk header and the corresponding in memory slot entries are updated. In the multiplecachedisk *(mcd)* configuration, a roundrobin algorithm is used to select the cache disk to receive data. All data are written to the cache disk(s) in "append" mode, which ensures that every time the data are written to consecutive cache disk blocks.

For *Level 2 destage*, we use a "lastwritefirstdestage" algorithm according to the LRU List. Each time a data segment (64Kb in our preliminary implementation, and it is a configurable parameter to make use of optimal striping unit size [CHE 95]) are read from the consecutive blocks of the cache disk and then written to the disk array in parallel. At this point, parity code is calculated and written to the parity block. After data is destaged to the data disk array, the LRU list and free list are updated subsequentially.

7.3.4 Reliability analysis

One potential reliability problem with this virtual NVRAM cache is that the cache disk fails before data are destaged to the data disk array. The cache disk may become a potential single point of failure. To address this problem, we can mirror the cache disk by using another partition on a different disk which acts as a backup cache disk. In this case the *Level 1 destage* writes data from the RAM buffer to cache disk and backup cache disk in parallel. Since the total size of a good performance cache is usually in the range of 1% of total data volume of the storage system, we would not expect significant waste in terms of disk space. On the other hand, the cost of disk is rapidly dropping and it is quite practical to have a one percentage point increase in disk space to trade for high reliability.

VCRAID uses immediate report mode for writes as discussed in section 7.3.1, except for those requests with the "*O_SYNC*" flags. This immediate report does not cause a serious reliability problem for the following reasons: First, the delay caused by the *VCRAID* driver is limited to several hundreds of milliseconds at most.

The driver writes the data to the cache disk within several hundreds of milliseconds. As default, the Linux caches file data and metadata in RAM for 30 and 5 seconds, respectively, before they are flushed into the disks. We believe that the additional several hundreds of milliseconds should not cause any problem. In fact, we can use a tracking structure to track the metadata dependency and adopt a mechanism similar to Soft Updates [MCK 99] to further protect the metadata integrity and consistency.

Second, the *VCRAID* RAM buffer is reserved during the system boot and is not controlled by the OS kernel. It is isolated from the other part of RAM used by the system and used exclusively by *VCRAID* driver. It should have less chance to be crashed by other applications. Finally, a small amount of NVRAM can be used in the real implementation to further guarantee the reliability before the data is written to the cache disks.

Our current implementation of *VCRAID* is for RAID level 5. As soon as data are destaged to the data disk array, they are parity protected in the same way as RAID 5.

Another issue about reliability is crash recovery. In case of system crash, data can be recovered from the cache disk. The system first switches to the single user mode. All data that have not destaged to the data disk array are written to data disk array according to the cache disk headers which record the relationships between a cache disk block and the corresponding data disk block. At this point the *VCRAID* returns to a clean and stable state. Then the file system can start the normal crashrecovery process by running "*e2fsck*".

7.4 Performance evaluations

7.4.1 Experimental setup

We installed and tested our drivers on a Gateway G6400 machine running Linux 2.0.36. The machine has 64MB DRAM, two IDE interfaces and two sym53c875 SCSI adapters. Six hard disks (including 4 SCSI disks and 2 IDE disks) are connected to this machine. The disk parameters are listed in Table 7.1. In the following benchmark tests, unless specified otherwise, the size of *VCRAID* RAM buffer is 4M. A cache disk size is 200MB with data block size being 1KB. The 4 SCSI disks are used to form the disk array while the OS kernel runs on the first IDE disk (model 91366U4). The second IDE disk (model 91020D6) is used as cache disk in the single cache disk *(scd)* configuration. For the case of the multiple cache disk*(mcd)* configuration, two logical partitions, one from each of the IDE disks, are used as cache disks.

Table 7.1: Disk parameters

Disk Model	Interface	Capacity	Data buffer	RPM	Latency (ms)	Transfer rate (MB/s)	Seek time (ms)
DNES-318350	Ultra SCSI	18.2G	2MB	7200	4.17	12.7-20.2	7.0
DNES-309170	Ultra SCSI	9.1G	2MB	7200	4.17	12.7-20.2	7.0
91366U4	ATA-5	13.6G	2MB	7200	4.18	Up to 33.7	9.0
91020D6	ATA-4	10.2G	256KB	5400	5.56	18.6	9.0
ST52160N	Ultra SCSI	2.1G	128KB	5400	5.56	10	11
ST32151N	SCSI-2	2.1G	256KB	5400	5.54	10	10.4

The performance of *VCRAID* is compared to that of the standard software RAID0 and RAID5 shipped with the Linux operating system. To ensure that the initial environments are the same for every test, we performed tests for each benchmark run in steps as follows:

1. Load the corresponding driver *VCRAID*, RAID0 or RAID5 (using "*insmod*");
2. Execute the benchmark program;
3. Unload the driver (using the command "*rmmod*");
4. Reboot the system and prepare for the next test.

7.4.2 Benchmarks

Workload plays a critical role in performance evaluations. We paid special attention in selecting workload for our performance testing. In order to give a realistic performance evaluation and comparison, we use real world benchmarks in our performance measurements. Benchmark programs that are used in our performance measurements are ServerBench which is a very popular benchmark for testing server performance, Bonnie which is a standard benchmark used for unix systems, PostMark which is a popular file system benchmark and some user application programs that are disk I/O bound such as *untar, copy*, and *remove*.

7.4.2.1 ServerBench 4.1

ServerBench [SER 2000] is a popular benchmark developed by ZD Inc. and has been used by many organizations including ZD Lab to measure the performance of application servers in a client/server environment. It consists of three main parts: a client program, a controller program and a server program. A server, a controller and several clients are needed to run server program, controller program and client program respectively. The aforementioned G6400 machine acts as the server in the test, and 5 PCs with Celeron 500MHZ CPU, 64MB DRAM acts as controller and clients which run Windows 98. All machines are equipped with 3C905B 10/100M network adaptors and interconnected through a DLink 10/100M 8 port switch. Our test suites are based on the standard test suite provided by ServerBench. The size of data set is 128 MB, which is divided into 4 data segments with each segment being 32MB.

The performance metric used in this measurement is *Transactions Per Second* (TPS) which is the standard performance measure as output of ServerBench. For each run, we measured the TPS results of standard software RAID0, RAID5, *scd* (Single Cache Disk) *VCRAID*, and *mcd* (Multiple Cache Disk) *VCRAID* under the same conditions. Within each run, there are

4 test sets corresponding to 1 to 4 clients respectively. For each test set, there are 16 groups (mixes), each with 32 transactions. Every transaction comprises 3 basic steps: client requests (read or write) to the server; server accepts and processes the request; and finally client receives a response from the server. The percentage of write requests, referred to as *write ratio*, is an important parameter to be determined in each test round. We determine the write ratio based on our observations of existing disk I/O traces and previously published data.

7.4.2.2 Bonnie

Bonnie [BON 96] is a benchmark used to measure the performance of Unix file system operations. It performs a series of tests on a file of known size. For each test, Bonnie reports the results of sequential output, sequential input and random seek time for both block device and character devices. Since disk driver and RAID driver are both block devices, in our experiments we are only concerned with performance of block devices. We measured write performance and random seek performance for block devices. For block writes, a file is created using system call *write()*. For random seeks, we run concurrent processes operating on one large file of 100MB. The 4 processes perform a total of 200,000 *lseek()s* to random locations computed using *random()*. Each such random seek is followed by a read operation using system call, *read()*. Among all these random read operations, 10% of them are rewritten immediately in the same location.

7.4.2.3 PostMark

PostMark [KAT 99] is a popular file system benchmark developed by Network Appliance. It measures system throughput in terms of transaction rates in an ephemeral smallfile environment by creating a large pool of continually changing files. "PostMark was created to simulate heavy smallfile system loads with a minimal amount of software and configuration effort and to provide complete reproducibility [KAT 99]". PostMark generates an initial pool of random text files ranging in size from a configurable low bound to a configurable high bound. This file pool is of configurable size and can be located on any accessible file system. Once the pool has been created, a specified number of transactions occur. Each transaction consists of a pair of smaller transactions, i.e. *Create file or Delete file* and *Read file or Append file*. Each transaction type and its affected files are chosen randomly. The read and write block size can be tuned. On completion of each run, a report is generated showing some metrics such as elapsed time, transaction rate, total number of files created and so on.

7.4.2.4 Untar/Copy/Remove

We have created a set of our own metadata intensive benchmark programs. This benchmark consists of 3 sets of script programs: *untar, copy*, and *remove. Untar* creates a directory tree by untaring a tar file (using "*tar xf*" command). The particular tar file we used is the standard test suites file of webbench by ZDNet. To make the data set larger, we double the test suites. The resulting file tree contains 952 subdirectories with different depths and 12322 files with an average file size being 9.9 KB. The total size of the directory tree is 122MB. This source directory tree resides in a local system disk. The *copy* and *remove* perform users' copying (using the "*cp r*" command) and removing (using the "*rm rf*" command) the entire directory tree created by the *untar* test. The performance metric used for this benchmark is user response time since in this case user would be mostly interested in response time. The smaller number means better performance.

7.4.3 Numerical results and discussions

With the experimental settings and the benchmarks described above, we have carried out extensive experiments to evaluate the performance of *VCRAID* and compare it with standard software RAID0 and RAID5 shipped with Linux package. All numerical results presented in this subsection are the average of 3 experimental runs. In our tests, **mcd** and **scd** stand for VCRAID with multiple cache disks and VCRAID with single cache disk, respectively.

7.4.3.1 Transactions Per Second (TPS)

Figure 7.4 shows the average transactions per second (TPS). the performance score output by ServerBench, as a function of number of clients. Three separate figures are plotted corresponding to different write ratios. As mentioned previously, the write ratios were selected based on our observation of typical disk I/O traces. For example, our observation of HP traces (Snake, Cello, and hplajw) [RUE 93] and EMC traces [HU 99] indicate that average write ratio in a typical office and engineering environment is about 57%. Hua et al [HUA 99] used write ratios of 0.1, 0.5 and 0.9 in their performance evaluation for cached RAID system. From Figure 7.4a, we can see that when the write ratio is very low (0.1) the performances of *VCRAID*, standard software RAID0 and RAID5 are very close. One can hardly notice the difference in terms of TPS for these different RAID systems implying that the software RAID performs fine with such low write ratio.

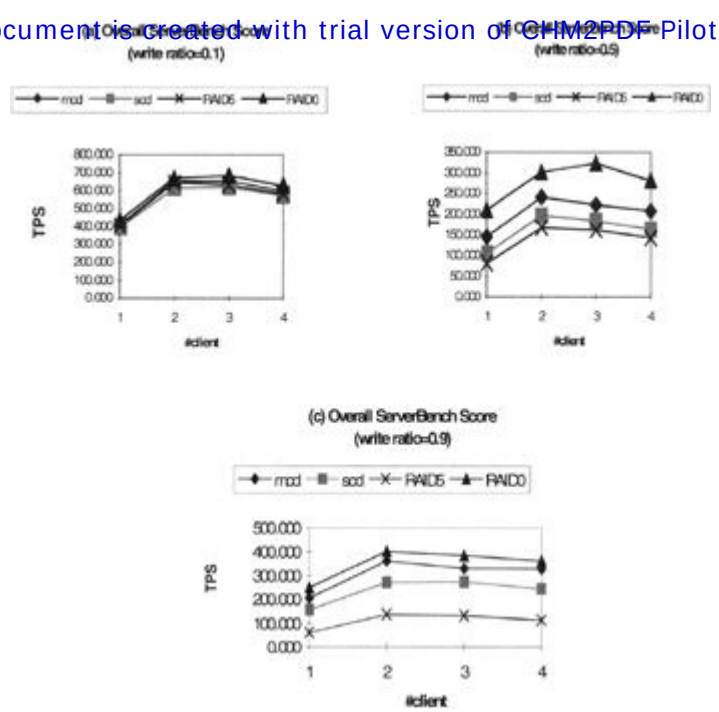


Figure 7.4: VCRAID vs. RAID5 and RAID 0 (Mean request size=lk)

As the system RAM size increases in today's server computers, a large portion of disk read operations are cached in the system RAM. That is, the large system RAM filtered out many disk read operations. As a result, the proportion of write operations as seen by the disk I/O system increases. When the write ratio increases, the performance penalty due to small write problems of RAID5 comes into the picture. Figures 7.4b and 7.4c show the TPS for write ratios being 0.5 and 0.9 respectively. For these types of workloads, our *VCRAID* shows significant performance advantages.

With multiple cache disks (*mcd*), the TPS number is doubled compared to standard software RAID5 for write ratio of 0.5 and more than tripled for write ratio of 0.9. We noticed in all our measurements that TPS drops after the number of clients exceeds 2. This result can be mainly attributed to the network congestion caused by NIC at the server. The 3C905B 10/100M network adapter at the server is not fast enough to handle a large number of requests from more clients.

In order to observe how much the *VCRAID* can hide the small write problem of RAID5, we also compared the performance of *VCRAID* to that of software RAID0. Since RAID0 has no redundancy, there is no overhead for write operations such as parity computations and parity updates. Figures 7.4b and 7.4c show that *mcd* can realize about 80% performance of RAID0 for write ratio of 0.5 and over 90% performance of RAID0 for write ratio of 0.9.

The reason for the performance gap between the *VCRAID* and RAID0 can be explained as follows. The workload to the server generated by ServerBench is fairly random with no spatial and temporal localities. As a result, destage operations may affect the overall performance because the system cannot find idle period to do destage operations.

Furthermore, some read operations may find data in cache disk in the *VCRAID* limiting the parallelism for these reads. This is why the performance gap is getting smaller as write ratio increases as shown in Figure 7.4c.

We noticed performance different between single cache disk (*scd*) *VCRAID* and multiple cache disk (*mcd*) *VCRAID* in Figure 7.4b and 7.4c. We believe that this difference mainly results from destage overhead. In order to verify this, we change the average request size issued from clients to the server from 1 KB to 0.5 KB. The measured results are shown in Figure 7.5. As is shown, with smaller request size, the performance difference between *scd* and *mcd* becomes smaller. This is because the frequency of destage operation is smaller due to small data sizes. For large request sizes, multiple cache disk allows destage operations to be performed in parallel resulting in better performance than single cache disk case.

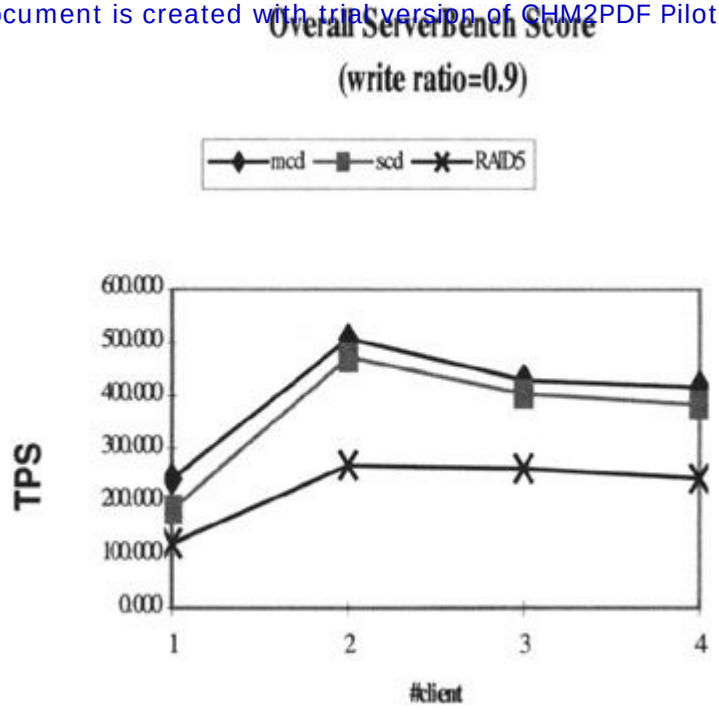


Figure 7.5: scd vs. mcd (mean request size=0.5k)

To examine the effect of the size of the RAM buffer reserved for *VCRAID* on the overall performance of the *VCRAID* system we plotted the TPS as a function of RAM buffer size as shown in Figure 7.6. Our measurement results show that 4 MB RAM buffer gives optimal performance. Smaller RAM may get filled up quicker than cache disk can absorb resulting in wait time for some I/O requests. If the RAM buffer is too large, on the other hand, the remaining system RAM used for a regular file system becomes small, adversely affecting the overall performance. Depending on the workload environment, the optimal RAM buffer size should be tuned at set up time.

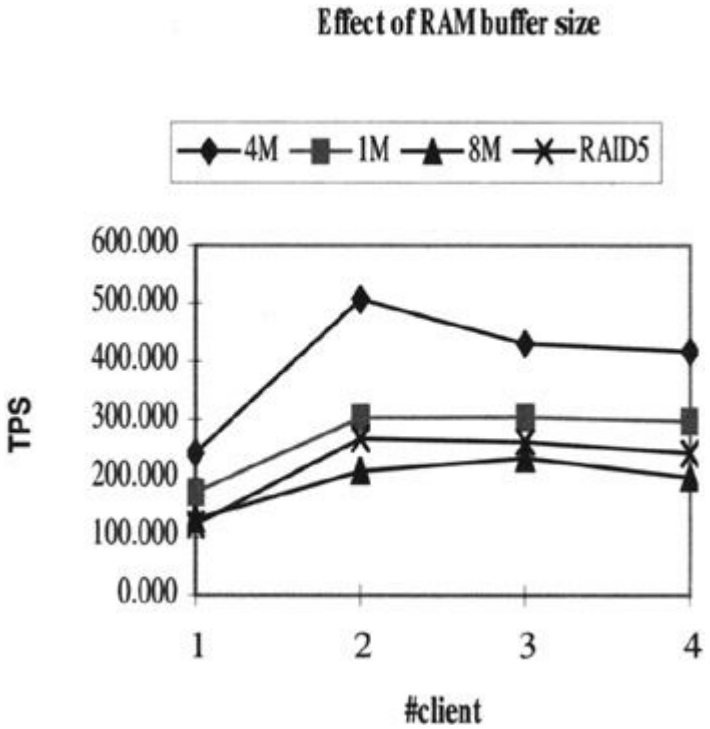


Figure 7.6: Effect of RAM buffer size

7.4.3.2 Response times

Our next experiment is to measure and evaluate user response times by running realistic application programs under Linux operating system such as *untar, remove*, and *copy*. Figure 7.7 shows user response times or execution times of standard software RAID0, RAID5, and *VCRAID* including *scd* and *mcd* for both synchronous and asynchronous mode.

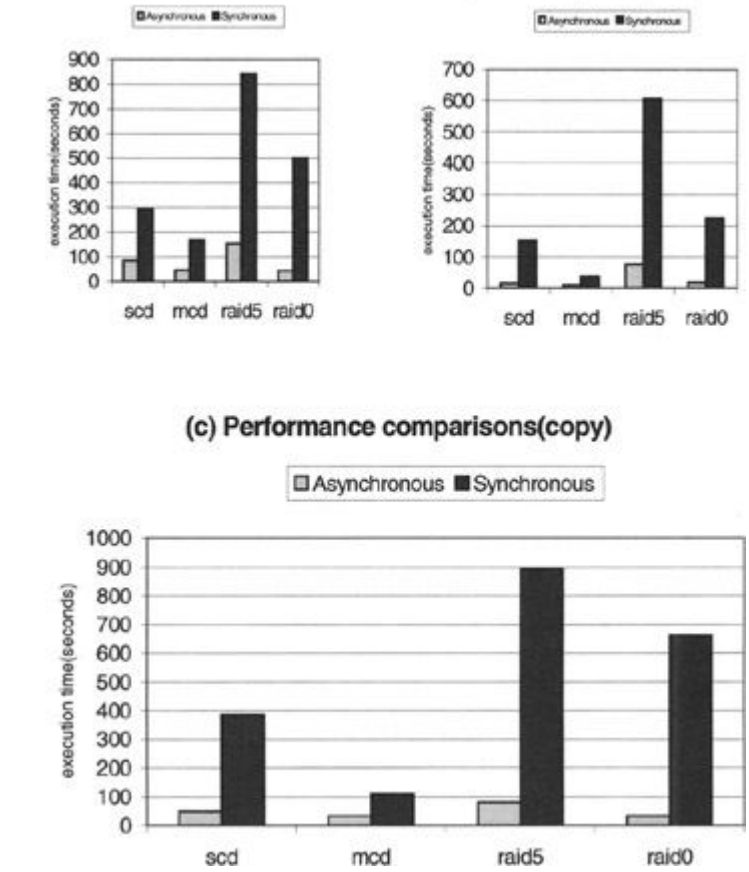


Figure 7.7: Results of untar/copy/remove

In the case of synchronous request mode, the performance gains of *mcd VCRAID* over RAID5 are a factor of 7.0 for untar (Figure 7.7a), a factor of 16 for remove (Figure 7.7b), and a factor of 4.2 for copy (Figure 7.7c) respectively. Even the *scd VCRAID* presents performance improvements of a factor 2.8, 3.9, and 2.3, respectively for the three programs. It can be seen from these figures that the *VCRAID* even performs better than standard RAID0.

There are three reasons for such a great performance gain. First of all, in this experiment we measured performance of real applications. As we indicated earlier, real applications have strong data locality properties. Cache works only when locality exists. This is true for any cache. A CPU cache would not work if application programs do not have locality properties and evenly access data across entire main memory. This is also true for disk caches that would work and show performance advantages if disk accesses have locality properties.

Secondly, there are many overwrites for these metadata intensive workloads such as modifying super block, group descriptors, directory entries. The *VCRAID* can capture these overwrite operations in the write cache before they go to disks. As a result, frequency of destage operations is reduced significantly in real applications. The final reason is that even with RAID0, small data are written to data disk under this synchronous request mode whereas burst simultaneous small requests are combined into large requests in *VCRAID*.

In the case of asynchronous requests, we also observed significant performance gains compared to standard software RAID. The performance improvements range from a factor of 2.5 (Figure 7.7c) to 7.9 (Figure 7.7b). This great performance gain can be mainly attributed to the effective and large cache size of the *VCRAID*. In our test, the total system RAM is 64MB. Taking into account the space used by OS, the available RAM for caching and buffering is about 50MB, 16MB of which is used for system buffering, so the RAM available for file cache is only about 34MB.

Even though user process does not have to wait until data are written into disk, it is often blocked because the file cache size is much smaller than data set of the experiment, which is 122MB. However, the cache size of *VCRAID* is 200MB including RAM and cache disk. This large hierarchical disk cache appears to the host as a large RAM that can absorb quickly a large amount of write requests resulting in great performance gain.

We also noticed that the performance of *VCRAID* is even much better than that of RAID0 for *remove* program for both synchronous and asynchronous mode (Figure 7.7b). The reason for this is as follows. When a file or a directory is removed, the super block, group descriptors, block and inode bitmap and corresponding inodes are updated while the "real data" are not removed. In our test, 952 subdirectories and 12322 files are removed, many blocks are overwritten over and over again. Most of these operations are small writes. *VCRAID* can cache these small writes effectively giving rise to great performance gain.

7.4.3.3 Bonnie throughput

Our next experiment is to compare the throughput of *VCRAID* to that of RAID0 and RAID5 using the benchmark program Bonnie. The size of the data set is 100M bytes. Figures 7.8a and 7.8b show the results for block write and random seek

respectively. The performance gains of *scd* and *mcd* over RAID5 are 2.4 and 3.65 for block write as shown in Figure 7.8a. We also notice that the performance of *mcd* are comparable to that of RAID0, because all requests are write operations under the block write test, where small writes are absorbed by Virtual NVRAM cache.

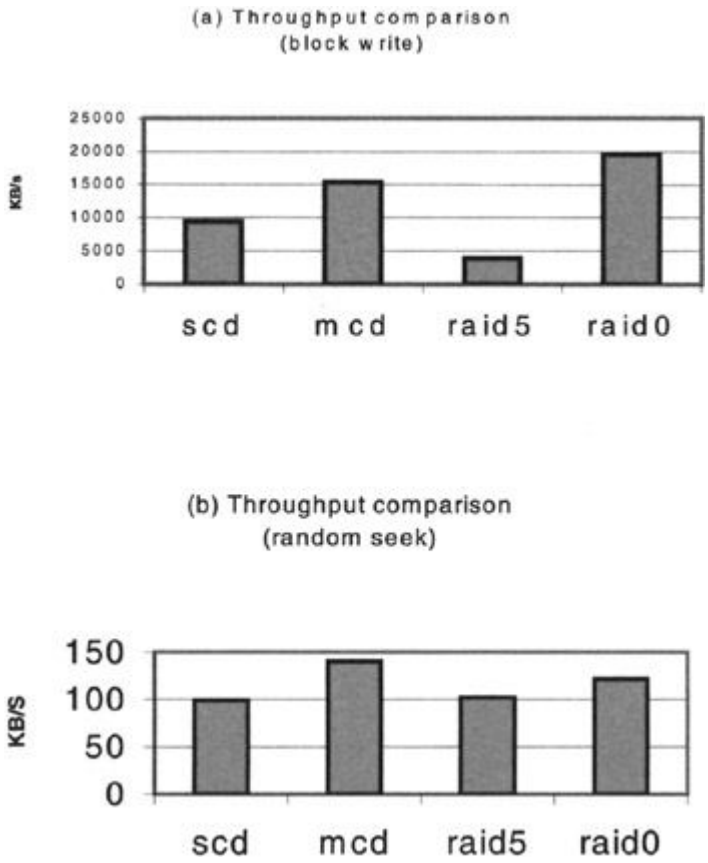


Figure 7.8: Results for Bonnie Benchmark

For the random seek operations, we observed very small performance gain of *VCRAID* over RAID5 as shown in Figure 7.8b. The reason is that 90% of requests are read requests during the random seek test, which are bypassed to the disk array by *VCRAID* directly. Furthermore the 10% follow up write operations are in place write with no seek time involved.

Therefore, the performance of RAID5, *VCRAID* and RAID0 are very close (Figure 7.8b). From Figure 7.8b we also noticed that the performance of random seeks are very poor, which confirms the claim of the Bonnie author, "random seeks on Unix file systems are appallingly slow [BON 96]".

7.4.3.4 PostMark throughput

Our final experiment is to use PostMark to measure the I/O throughput in terms of transactions per second. PostMark measures performance in terms of transaction rates in the ephemeral smallfile regime by creating a large pool of continually changing files.

The file pool is of configurable size. In our tests, PostMark was configured in three different ways as in [KAT 99], i.e: 1) small: 1000 intial files and 50000 transactions; 2) medium: 20000 initial files and 50000 transactions; and 3) large: 20000 initial files and 100000 transactions.

The read and block sizes are set to 1KB which is the default block size on Linux. We left all other PostMark at their default settings. Our *VCRAID* was configured by using single cache disk (*scd*). The results of testing are shown in Table 7.2, where larger numbers indicate better performance.

Table 7.2: PostMark results in terms of transactions per second

Series	RAID0	SCD	RAID5
Small	1111	941	561
Medium	68	63	30
Large	31	28	16

From these results, we see that *VCRAID* exceeds the throughput of the built in software RAID 5 and is comparable to that of RAID 0. The performance improvement of *VCRAID* over RAID 5 ranges from 67.7% to 110%.

7.5 Related work

A number of approaches for overcoming small write problems in RAID5 exist in the literature.

Nonvolatile RAM (NVRAM). Modern hardware RAID systems make extensive use of NVRAM write caches to allow fast write [CHE 94][HOU 97][MEN 93][TRE 95], or asynchronous write. Write requests are acknowledged before they go to disk. Such write caches significantly reduce user response times of RAID. Large write caches can also improve system throughput by taking advantages of both temporal locality and spatial locality of general workloads. While most commercial hardware RAID systems have large write caches for better performance, there is no readily cache solutions for software RAID. *VCRAID* uses a small RAM and a log disk to form a large and nonvolatile write cache.

Striping. A scheme called *parity striping* [GRA 90] proposed by Jim Gray et al has similar performance to a RAID with infinitely large striping unit. Chen and Lee [CHE 95] investigate how to stripe data across a RAID Level 5 disk array for various workloads and studies the optimal striping unit size for different numbers of disks in an array. Jin et al [JIN 98] divided the stripe write operation in RAID 5 into two categories, *full stripe write* and *partial stripe write*. They proposed an adaptive control algorithm to improve the partial stripe write performance by reducing the stripe read/write operations. Mogi and Kitsuregawa proposed a *dynamic parity stripe reorganization* [MOG 94] that creates a new stripe for a group of small writes and hence eliminating extra disk operations for small writes all together. Hua, Vu and Hu improve the *dynamic parity stripe reorganization* further by adding distributed buffer for each disk of RAID [HUA 99]. *AFRAID* [SAV 96] proposed by Savage and Wilkes eliminates the small write problem by delaying parity updates and allowing some stripes to be nonredundant for a controllable period of time.

Logging. Logging techniques are used to cure the small write problem in many researches [CHE 2000][GAB 98][SEL 93][SHI 95][STO 93]. A very intelligent approach called *Parity Logging* [STO 93] proposed by Stodolsky, Holland and Gibson makes use of high speed of large disk access to log parity updates in a log disk. As a result, many parity changes are collected and are written into disk in large sizes, read from disk in large sizes, and compute new parity in large sizes. Another approach is *Data Logging* proposed by Gabber and Korth [GAB 98]. Instead of logging parity changes, it logs the old data and new data of a small write in a log disk and compute parity at later time. Data logging requires 3 disk operations for each small write: reading old data, writing new data in place, and writing a log of old data and new data in log disk, as opposed to 4 operations in RAID5.

Zebra [HAR 95] and *xFS* [AND 95] are both distributed file systems that use striped logging to store data on a collection of servers. *Zebra* combines LFS with RAID to allow for faster disk operation, and a single metadata server is used. *xFS* eliminates the centralized elements of Zebra that could cause a bottleneck. Both *Zebra* and *xFS* are complete file systems, while our *VCRAID* is a device level driver.

HP *AutoRAID* [WIL 95] uses a two level storage hierarchy and combines the performance advantages of mirroring with the cost capacity benefits of RAID5 by mirroring active data and storing inactive data in RAID5. If the active subset of data changes relatively slowly over time, this method has great performance/cost benefits. Similarly, another scheme called *dynamic parity grouping* (DPG) proposed by Yu et al [YU 2000] partitions the parity into special parity group and default parity group. Only the special parity blocks are buffered in the disk controller cache, while the default parity blocks remain on disks.

7.6 Concluding remarks

We have presented the design and implementation of a virtual NVRAM cache (*VCRAID*) for software RAID. We called it virtual NVRAM because it does not have real or physical NVRAM, nor does it need any additional hardware. The virtual NVRAM cache is completely implemented using software while keeping properties of NVRAM. It is nonvolatile because the cache is part of a disk and it is fast due to the use of log structure in cache disks.

A prototype do it yourself RAID with a large virtual NVRAM cache (*VCRAID*) has been implemented under the Linux kernel. Through different benchmark tests, *VCRAID* demonstrates superb performance advantages over the standard built in software RAID5 shipped with the Linux package. Depending on the workload characteristics, performance gains due to *VCRAID* range from 67.7% to an order of magnitude. For applications that have data locality, we observed up to a factor of 16 performance improvements in terms of user response time. In many situations, *VCRAID* achieves similar performance as RAID0 and sometimes better than RAID0 indicating that *VCRAID* realizes close to maximum potential to hide small write problems in RAID5.

We are currently working on building a *VCRAID* with mirrored cache disks to further improve the reliability. We will also be working on porting *VCRAID* on hardware RAID controllers, where *VCRAID* will make use of controllers' RAM other than the host system RAM.

Acknowledgements

This research is supported in part by National Science Foundation under grants MIP9714370 and CCR0073377. The authors would like to thank the anonymous reviewers for their many helpful comments and suggestions.

7.7 Bibliography

[AND 95] Anderson T., Dahlin M., Neefe J., Patterson D., Roselli D., and Wang R., *"Serverless Network File Systems"*, In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, Dec. 36, 1995, p. 109126.

[ASA 95] Asami S., Talagala N., Anderson T., Lutz K., and Patterson D., *"The Design of LargeScale, DoItYourself RAIDs"*, available at URL: http://www.cs.berkeley.edu/pattrsn/papers.html , 1995.

[BON 96] Bonnie Benchmark, *"Bonnie benchmarks v2.0.6"*, 1996, available at URL: http://www.textuality.com/bonnie/ .

[CHE 94] Chen P., Lee E., Gibson G., Katz R., and Patterson D., *"RAID: HighPerformance, Reliable Secondary Storage"*, *ACM Computing Surveys* 26(2), 1994, p. 145-185.

[CHE 95] Chen P., and Lee E., *"Striping in a RAID Level 5 Disk Array"*, *Proceedings of the 1995 ACM SIGMETRICS joint international conference on Measurement and modelling of computer systems*, May 1519, 1995, p. 136-145.

[CHE 2000] Chen Y., Hsu W., and Young H., *"Logging RAID An Approach to Fast, Reliable, and LowCost Disk Arrays"*, *EuroPar'* 2000, p. 1302-1312.

[CHU92] Chutani S., Anderson O., Kazar M., Leverett B., Mason W., and Sidebotham R., *"The Episode File System"*, *Winter USENIX Conference*, Jan. 1992, p. 43-60.

[COO 96] Coombs D., *"Drawing up a new RAID roadmap"*, *Data Storage*, vol.3, Dec. 1996, p.59-61.

[GAB 98] Gabber E., and Korth H., *"Data Logging: A Method for Efficient Data Updates in Constantly Active RAIDs"*, *Proc. of the 14th Intl. Conference on Data Engineering*, 1998, p. 144-153.

[GRA 90] Gray J., Horst B., and Walker M., *"Parity Striping of Disc Arrays: LowCost Reliable Storage with Acceptable Throughput"*, *In Proc. Of the 16th Very Large Database Conference (VLDB)*, 1990, p.148-160.

[GRA 93] Gray J., and Reuter A., *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers, 1993.

[HAR 95] Hartman J., and Ousterhout J., *"The Zebra Striped Network File System"*, *ACM Transactions on Computer Systems*, vol.13 no.3, 1995, p.274-310.

[HOU 97] Hou R., and Patt Y., *"Using NonVolatile Storage to Improve the Reliability of RAID5 Disk Arrays"*, *the 27th Annual Intl. Symposium on Fault Tolerant Computing*, 1997, p.206-215.

[HU 96] Hu Y., and Yang Q., *"DCDdisk caching disk: A New Approach for Boosting I/O Performance"*, *23rd Annual Intl. Symposium on Computer Architecture*, Philadelphia, PA, May, 1996, p.169-178.

[HU 99] Hu Y., Yang Q., and Nightingale T., *"RAPID Cache: A Reliable and Inexpensive Write Cache for Disk I/O Systems"*, *Proc. of the 5th Intl. Symposium on High Performance Computer Architecture*, Jan. 1999, p. 204-213.

[HUA 99] Hua K., Vu K., and Hu T., *"Improving RAID Perfromance Using a Multibuffer Technique"*, *Proc. of the 15th Intl. Conference on Data Engineering*, 1999, p. 79-86.

[JIN 98] Jin H., Zhou X., Feng D., and Zhang J., *"Improving Partial Stripe Write Performance in RAID Level 5"*, *Proceedings of 2nd IEEE International Caracas Conference on Devices, Circuits and Systems*, March 24, 1998, Margarita, Venezuela, p.396-400.

[KAT99] Katcher J., *"PostMark: A New File System Benchmark"*, Technical Report TR3022, Network Appliance, http://www.netapp.com/tech_library/3022.html, 1999.

[MAT 97] Matthews J., Roselli D., Costello A., Wang R., and Anderson T., *"Improving the Performance of LogStructured File Systems with Adaptive Methods"*, *Proc. Sixteenth ACM Symposium on Operating System Principles*, Saint Malo, France, p5-8, October 1997.

[MCK 99] Mckusick M., and Ganger G., *"Soft Updates: A Technique for Eliminating Most Synchronous Writes in the*

Fast Filesystem", Proc. Of the FREENIX Track. 1999 USENIX Annual Technical Conference, p6-11, June 1999.

[MEN 93] Menon J., and Cortney J., *"The Architecture of a FaultTolerant Cached RAID Controller"*, *Proc. of the 20th Annual Intl. Symposium on Computer Architecture*, May 1993, p. 76-86.

[MOG94] Mogi K., and Kitsuregawa M., *"Dynamic parity stripe reorganization for RAID5 disk arrays"*, *Proc. Of the 3rd Intl. Conference on Parallel and Distributed Information Systems*, Sept. 1994, p. 17-26.

[MOH 95] Mohan C., *"Disk ReadWrite Optimizations and Data Integrity in Transaction Systems Using WriteAhead Logging"*, *Proc. 11th International Conference on Data Engineering*, March 1995.

[RUE 93] Ruemmler C., and Wilkes J., *"UNIX Disk Access Patterns"*, *Proc. of the USENIX'93 Winter Conference*, San Diego, CA, Jan. 1993, p. 405-420.

[SAV 96] Savage S., and Wilkes J., *"AFRAID A Frequently Redundant Array of Independent Disks"*, *Proc. of 1996 USENIX Annual Technical Conference*, Jan. 1996, p. 2226-2739.

[SEL 93] Seltzer M., Bostic K., Mckusick M., and Staelin C., *"An Implementation of a LogStructured File System for UNIX"*, *Winter USENIX Proceedings*, Jan. 1993, p. 201-220.

[SER 2000] SERVER BENCH 4.1, July 2000, URL: http://www.zdnet.co.uk/pcmag/labs/2000/07/os/22.html .

[SHI 95] Shirriff K., *"Sawmill: A Logging File System for a HighPerformance RAID Disk Array"*, *Technical Report, CSD95862*, Computer science department, University of California at Berkeley, 1995.

[STO 93] Stodolsky D., Holland M., and Gibson G., *"Parity Logging: Overcoming the small write Problem in Redundant Disk Arrays"*, *Proc. of the 21th Annual Intl. Symposium on Computer Architecture*, 1993, p. 64-75.

[TRE 95] Treiber K. and Menon J., *"Simulation study of cached RAID5 designs"*, *Proc. Of Intl. Symposium on High Performance Computer Architectures*, Jan. 1995, p. 186-197.

[WIL 95] Wilkes J., Golding R., Staelin C. and Sullivan T., *"The HP AutoRAID Hierarchical Storage System"*, *Proc. Of the Fifteenth ACM Symposium on Operating System Principles*, Dec. 36, 1995, p. 96-108.

[YU2000] Yu P., Wu K., and Dan A., *"Dynamic Parity Grouping for Efficient Parity Buffering for RAID5 Disk Arrays"*, *Computer Systems Science and Engineering*, vol. 15 No. 3, May 2000, p. 155-163.

Part Three: Appendices

Appendix List

Appendix 1: Matrix Product MPI-2 Codes

Christophe Cérin, Université de Picardie Jules Verne, LaRIA, 5 rue du moulin neuf, 80000 Amiens, France
Hai Jin, Huazhong University of Science and Technology, Wuhan, 430074, China

Description of useful MPI I/O primitives

We suppose the reader to be familiar with rudiments of MPI. We comment only on the functions of I/O data management of the codes that follow. The description of instructions comes from http://www.mpi-forum.org. Please also refer to http://www.mpi-forum.org/docs/mpi-20-htm1/node171.htm#Node171 for the complete documentation about I/Os: some concepts have not been covered in the book, so reader is invited to read the whole of the documentation before programming suggested exercices.

MPI_Type_struct: the semantics of this operation as it is given by the MPI standard is:

```
int MPI_Type_struct( count, blocklens, indices, old_types, newtype )
int           count;
int           blocklens[];
MPI_Aint      indices [];
MPI_Datatype  old_types[];
MPI_Datatype *newtype;
Input Parameters

count - number of blocks (integer) -- also number of entries in arrays
array_of_types, array_of_displacements and array_of_blocklengths

blocklens - number of elements in each block (array)

indices - byte displacement of each block (array)

old_types - type of elements in each block (array of handles to
datatype objects)

Output Parameter

newtype - new datatype (handle)
```

MPI_FILE_SET_VIEW: the semantics of this operation as it is given by the MPI standard is:

```
MPI_FILE_SET_VIEW(fh, disp, etype, filetype, datarep, info)
[ INOUT fh] file handle (handle)
[ IN disp] displacement (integer)
[ IN etype] elementary datatype (handle)
[ IN filetype] filetype (handle)
[ IN datarep] data representation (string)
[ IN info] info object (handle)


The MPI_FILE_SET_VIEW routine changes the process's view of the data
in the file. The start of the view is set to disp; the type of data is
set to etype; the distribution of data to processes is set to
filetype; and the representation of data in the file is set to
datarep. In addition, MPI_FILE_SET_VIEW resets the individual file
pointers and the shared file pointer to zero. MPI_FILE_SET_VIEW is
collective; the values for datarep and the extents of etype in the
file data representation must be identical on all processes in the
group; values for disp, filetype, and info may vary. The datatypes
passed in etype and filetype must be committed.

The etype always specifies the data layout in the file. If etype is a
portable datatype (see Section Semantic Terms), the extent of etype is
computed by scaling any displacements in the datatype to match the
```

file data representation. If etype is not a portable datatype, no scaling is done when computing the extent of etype. The user must be careful when using nonportable etypes in heterogeneous environments; see Section Datatypes for File Interoperability for further details.

MPI_File_read_at_all: the semantics of this operation which allow explicit displacements to access data as it is given by the MPI standard is:

```
MPI_FILE_READ_AT(fh, offset, buf, count, datatype, status)
[ IN fh] file handle (handle)
[ IN offset] file offset (integer)
[ OUT buf] initial address of buffer (choice)
[ IN count] number of elements in buffer (integer)
[ IN datatype] datatype of each buffer element (handle)
[ OUT status] status object (Status)

MPI_FILE_READ_AT_ALL(fh, offset, buf, count, datatype, status)
[ IN fh] file handle (handle)
[ IN offset] file offset (integer)
[ OUT buf] initial address of buffer (choice)
[ IN count] number of elements in buffer (integer)
[ IN datatype] datatype of each buffer element (handle)
[ OUT status] status object (Status)

MPI_FILE_READ_AT_ALL is a collective version of the blocking
MPI_FILE_READ_AT interface.

MPI_FILE_WRITE_AT(fh, offset, buf, count, datatype, status)
[ INOUT fh] file handle (handle)
[ IN offset] file offset (integer)
[ IN buf] initial address of buffer (choice)
[ IN count] number of elements in buffer (integer)
[ IN datatype] datatype of each buffer element (handle)
[ OUT status] status object (Status)
```

MPI_File_write_at_all: (replace the "read" term by "write" in the previous definition)

MPI_Type_create_subarray: Subarray Datatype Constructor.

```
MPI_TYPE_CREATE_SUBARRAY(ndims, array_of_sizes, array_of_subsizes,
                         array_of_starts, order, oldtype, newtype)
In ndims number of array dimensions (positive integer)
IN array_of_sizes number of elements of type oldtype in each
   dimension of the full array (array of positive integers)
IN array_of_subsizes number of elements of type oldtype in each
   dimension of the subarray (array of positive integers)
IN array_of_starts starting coordinates of the subarray in each
   dimension (array of nonnegative integers)
IN order array storage order flag (state)
IN oldtype array element datatype (handle)
OUT newtype new datatype (handle)
The subarray type constructor creates an MPI datatype describing an
n-dimensional subarray of an n-dimensional array. The subarray may be
situated anywhere within the full array, and may be of any nonzero size
up to the size of the larger array as long as it is confined within
this array. This type constructor facilitates creating filetypes to
access arrays distributed in blocks among processes to a single file
that contains the global array.

This type constructor can handle arrays with an arbitrary number of
dimensions and works for both C and Fortran ordered matrices (i.e.,
row-major or column-major). Note that a C program may use Fortran order
and a Fortran program may use C order.

The ndims parameter specifies the number of dimensions in the full data
array and gives the number of elements in array_of_sizes,
array_of_subsizes, and array_of_starts.

The number of elements of type oldtype in each dimension of the
n-dimensional array and the requested subarray are specified by
```

array_of_sizes and array_of_subsizes, respectively. For any dimension i, it is erroneous to specify array_of_subsizes[i] < 1 or array_of_subsizes[i] > array_of_sizes[i].

The array_of_starts contains the starting coordinates of each dimension of the subarray. Arrays are assumed to be indexed starting from zero. For any dimension i, it is erroneous to specify array_of_starts[i] < 0 or array_of_starts[i] > (array_of_sizes[i] - array_of_subsizes[i]).

General framework

```
/*
 * MPI-2: Out-of-Core Matrix Product - Framework of the MPI-2 implementation
 * http://www.pdc.kth.se/training/Talks/SMP/maui98/jpprost/ex2_c_incomp.htm
 *
 * Author: Jean-Pierre Prost
 * Email: jpprost$$us.ibm.com
 * Home Page: http://www.research.ibm.com/people/p/prost
*/

#define P           4             /* number of tasks       */
#define Nb          64            /* slice size            */
#define Ng          (Nb * p)      /* global size of matrix */

. . . .
colext = Nb * extent;
slicelen = Ng * Nb;
sliceext = slicelen * extent;

/* create filetype for matrix A */
/* TO BE COMPLETED */

/* create filetype for matrix B */
/* TO BE COMPLETED */

/* create filetype for matrix C */
/* TO BE COMPLETED */

/* open files */
/* TO BE COMPLETED */

/* set views */
/* TO BE COMPLETED */

/* read horizontal slice of A */
/* TO BE COMPLETED */

/* loop on vertical slices */
for ( iv = 0; iv < P; iv++ ) {

  /* read next vertical slice of B */
  /* TO BE COMPLETED */

  /* compute block product */
  for ( i = 0; i < Nb; i++ ) {
    for ( j = 0; j < Nb; j++ ) {
      val_C[ i ][ j ] =0;
      for ( k = 0; k < Ng; k++ ) {
        val_C[ i ][ j ] += val_A[ i ][ k ] * val_B[ k ][ j ];
      }
    }
  }

  /* write block of C */
  /* TO BE COMPLETED */

}

/* close files */
/* TO BE COMPLETED */
```

A new code for the matrix product

We compute the product of two squared matrices

$$C = A * B = \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{24} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} * \begin{bmatrix} B_{11} & B_{12} & B_{13} & B_{14} \\ B_{21} & B_{22} & B_{23} & B_{24} \\ B_{31} & B_{32} & B_{33} & B_{24} \\ B_{41} & B_{42} & B_{43} & B_{44} \end{bmatrix}$$

according to the physical data representation for the file containing matrix A:

$A_{11}, A_{12}, A_{13}, A_{14}, A_{21}, A_{22}, A_{23}, A_{24}, A_{31}, A_{32}, A_{33}, A_{24}, A_{41}, A_{42}, A_{43}, A_{44}$

and for the file containing matrix B:

$B_{11}, B_{21}, B_{31}, B_{41}, B_{12}, B_{22}, B_{32}, B_{42}, B_{13}, B_{23}, B_{33}, B_{43}, B_{14}, B_{24}, B_{24}, B_{44}$

Example:

```
Matrix A:
 00001 00002 00003 00004 00005 00006 00007 00008
 00009 00010 00011 00012 00013 00014 00015 00016
 00017 00018 00019 00020 00021 00022 00023 00024
 00025 00026 00027 00028 00029 00030 00031 00032
 00033 00034 00035 00036 00037 00038 00039 00040
 00041 00042 00043 00044 00045 00046 00047 00048
 00049 00050 00051 00052 00053 00054 00055 00056
 00057 00058 00059 00060 00061 00062 00063 00064
Matrix B:
 00001 00009 00017 00025 00033 00041 00049 00057
 00002 00010 00018 00026 00034 00042 00050 00058
 00003 00011 00019 00027 00035 00043 00051 00059
 00004 00012 00020 00028 00036 00044 00052 00060
 00005 00013 00021 00029 00037 00045 00053 00061
 00006 00014 00022 00030 00038 00046 00054 00062
 00007 00015 00023 00031 00039 00047 00055 00063
 00008 00016 00024 00032 00040 00048 00056 00064
Matrix C (product):
 00204 00492 00780 01068 01356 01644 01932 02220
 00492 01292 02092 02892 03692 04492 05292 06092
 00780 02092 03404 04716 06028 07340 08652 09964
 01068 02892 04716 06540 08364 10188 12012 13836
 01356 03692 06028 08364 10700 13036 15372 17708
 01644 04492 07340 10188 13036 15884 18732 21580
 01932 05292 08652 12012 15372 18732 22092 25452
 02220 06092 09964 13836 17708 21580 25452 29324
```

The proposed code is as follows:

```
/*
/usr/local/mpi/bin/mpicc -0 out-of-core-matrix.c
*/

#include "sys/types.h"
#include "stdlib.h"
#include "stdio.h"
#include "string.h"
#include <argz.h>
#include <stdlib.h>
#include <limits.h>
#include <string.h>
#include <math.h>
#include <sys/time.h>
#include <sys/stat.h>
#include <unistd.h>
#include "mpi.h"
```

```
#define MPI_OFFSET_ZERO 0

#define P          4          /* number of tasks                 */
#define Nb         8          /* slice size. Nb*Nb = matrix size */
#define Ng         (Nb * P )  /* global size of matrix           */

static char fileA[] = "fileA";
static char fileB[] = "fileB";
static char fileC[] = "fileC";

void
fill_file_A (char *FNAME, int N)
{
  int              i, data;
  FILE             *f;

  /*  sprintf (s, FNAME, 0);*/
  if ((f = (FILE *) fopen(FNAME, "w+")) == NULL) {
    perror("io2: error in opening auxiliary files");
    exit(1);
  }
  for (i = 0; i < N * N; i++) {
    data = i + 1;
    if (fwrite (&data, sizeof(int), 1, f) != 1) {
      perror("io3: can't write in an auxiliary file");
      fclose(f);
      exit(1);
    }
  }
  fclose(f);
}

void
fill_file_B(char *FNAME, int N)
{
  int             i, j, data;
  FILE            *f;

  /* sprintf(s, FNAME, 0;*/
  if ((f = (FILE *) fopen(FNAME, "w+")) == NULL) {
    perror("io2: error in opening auxiliary files");
    exit(1);
  }
  for (i = 0; i < N; i++) {
    for(j=0; j < N; j++) {
    data = (i+1) + N*j;
    if (fwrite(&data, sizeof(int), 1, f) != 1) {
      perror("io3: can't write in an auxiliary file");
      fclose(f);
      exit(1);
    }
    }
  }
  fclose(f);
}

/*
 * Print screen fname file (which much contain integers)
 */
void
aff(char *fname, int N)
{
  FILE           *fp;
  int             i, count=0;

  fp = fopen(fname, "r");
  fread(&i, 4, 1, fp);
```

```
    while (0 == feof(fp)) {
      printf("%6.5d ", i);
      if(((count+1) % N) == 0) {
        count = 0; printf("\n");
      } else count++;
      fread(&i, 4, 1, fp);
    }
    printf("\n"); fclose(fp);
  }


  int
  main( int argc, char *argv[] )
  {
    int i, j, k;
    int iv;
    int myrank, commsize;
    int extent;
    int colext;
    int slicelen;
    int sliceext;
    int length[ 3 ];
    MPI_Aint disp[ 3 ];
    MPI_Datatype type[ 3 ];
    MPI_Datatype ftypeA;
    MPI_Datatype ftypeB;
    MPI_Datatype ftypeC;
    MPI_Datatype slice_typeB;
    MPI_Datatype block_typeC;
    MPI_Datatype slice_typeC;
    int mode;
    MPI_File fh_A, fh_B, fh_C;
    MPI_Offset offsetA;
    MPI_Offset offsetB;
    MPI_Offset offsetC;
    MPI_Status status;
    int val_A[ Nb ] [ Ng ], val_B[ Ng ] [ Nb ], val_C[ Nb ] [ Nb ];

    /* initialize MPI */
    MPI_Init( &argc, &argv );
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank );
    MPI_Comm_size(MPI_COMM_WORLD, &commsize );

    /* We fille and we print the file contain */
    if(myrank == 0) {
      fill_file_A(fileA,Nb);
      fill_file_B(fileB,Nb);
      printf("File A:\n");
      aff(fileA,Nb);
      printf("File B:\n");
      aff(fileB,Nb);
    }
    /* exit(0);*/
    /*****************************/

    MPI_Type_extent(MPI_INT, &extent );
    colext = Nb * extent;
    slicelen = Ng * Nb;
    sliceext = slicelen * extent;

    /* create filetype for matrix A */
    length[ 0 ] = 1;
    length[ 1 ] = slicelen;
    length[ 2 ] = 1;
    disp[ 0 ] = 0;
    disp[ 1 ] = sliceext * myrank;
    disp[ 2 ] = sliceext * P;
    type[ 0 ] = MPI_LB;
```

```
    type[ 1 ] = MPI_INT;
    type[ 2 ] = MPI_UB;
    MPI_Type_struct( 3, length, disp, type, &ftypeA );
    MPI_Type_commit( &ftypeA );

    /* create filetype for matrix B */
    length[ 0 ] = 1;
    length[ 1 ] = slicelen;
    length[ 2 ] = 1;
    disp[ 0 ] = 0;
    disp[ 1 ] = sliceext * myrank;
    disp[ 2 ] = sliceext * P;
    type[ 0 ] = MPI_LB;
    type[ 1 ] = MPI_INT;
    type[ 2 ] = MPI_UB;
    MPI_Type_struct( 3, length, disp, type, &ftypeB );
    MPI_Type_commit( &ftypeB );

    /* create filetype for matrix C */
    MPI_Type_vector( Nb, Nb, Ng, MPI_INT, &block_typeC );
    MPI_Type_havector( P, 1, colext, block_typeC, &slice_typeC );
    length[ 0 ] = 1;
    length[ 1 ] = 1;
    length[ 2 ] = 1;
    disp[ 0 ] = 0;
    disp[ 1 ] = sliceext * myrank;
    disp[ 2 ] = sliceext * P;
    type[ 0 ] = MPI_LB;
    type[ 1 ] = slice_typeC;
    type[ 2 ] = MPI_UB;
    MPI_Type_struct( 3, length, disp, type, &ftypeC );
    MPI_Type_commit( &ftypeC );

    /* open files */
    mode = MPI_MODE_RDONLY;
    MPI_File_open( MPI_COMM_WORLD, fileA, mode, MPI_INFO_NULL, &fh_A );
    mode = MPI_MODE_RDONLY;
    MPI_File_open( MPI_COMM_WORLD, fileB, mode, MPI_INFO_NULL, &fh_B );
    mode = MPI_MODE_CREATE | MPI_MODE_WRONLY;
    MPI_File_open( MPI_COMM_WORLD, fileC, mode, MPI_INFO_NULL, &fh_C );

    /* set views */
    MPI_File_set_view (fh_A, MPI_OFFSET_ZERO, MPI_INT, ftypeA,
                       "native", MPI_INFO_NULL );
    MPI_File_set_view( fh_B, MPI_OFFSET_ZERO, MPI_INT, ftypeB,
                       "native", MPI_INFO_NULL );
    MPI_File_set_view( fh_C, MPI_OFFSET_ZERO, MPI_INT, ftypeC,
                       "native", MPI_INFO_NULL );

    offsetA = MPI_OFFSET_ZERO;
    offsetB = MPI_OFFSET_ZERO;
    offsetC = MPI_OFFSET_ZERO;

    /* read horizontal slice of A */
    MPI_File_read_at_all( fh_A, offsetA, val_A, slicelen, MPI_INT, &status );

    /* loop on vertical slices */
    for ( iv = 0; iv < P; iv++ ) {

    /* read next vertical slice of B */
    MPI_File_read_at_all( fh_B, offsetB, val_B, slicelen, MPI_INT, &status );
    offsetB += slicelen;

    /* compute block product */
    for (i = 0; i < Nb; i++ ) {
      for ( j = 0; j < Nb; j++ ) {
        val_C[ i ] [ j ] = 0;
        for ( k = 0; k < Ng; k++ ) {
```

```
            val_C[ i ] [ j ] += val_A[ i ] [ k ] * val_B[ k ] [ j ];
        }
      }
    }

    /* write block of C */
    MPI_File_write_at_all( fh_C, offsetC, val_C, Nb * Nb, MPI_INT, &status );
    offsetC += Nb * Nb;
  }

    /* close files */
    MPI_File_close ( &fh_A );
    MPI_File_close ( &fh_B );
    MPI_File_close ( &fh_C );

    /* free filetypes */
    MPI_Type_free( &ftypeA );
    MPI_Type_free( &ftypeB );
    MPI_Type_free( &ftypeC );

    if(myrank == 0) {
      printf("File C:\n");
      aff(fileC,Nb);
    }

    /* finalize MPI */
    MPI_Finalize();
}
```

An example of an execution is given below. Note that only process 0 prints its result. Note also what is contained in file C at the end of the execution (is it possible to diminish the size of file C?).

```
[cerin@fatboy cerin]$ /usr/local/mpi/bin/mpirun -np 4 a.out
File A:
 00001 00002 00003 00004 00005 00006 00007 00008
 00009 00010 00011 00012 00013 00014 00015 00016
 00017 00018 00019 00020 00021 00022 00023 00024
 00025 00026 00027 00028 00029 00030 00031 00032
 00033 00034 00035 00036 00037 00038 00039 00040
 00041 00042 00043 00044 00045 00046 00047 00048
 00049 00050 00051 00052 00053 00054 00055 00056
 00057 00058 00059 00060 00061 00062 00063 00064
File B:
 00001 00009 00017 00025 00033 00041 00049 00057
 00002 00010 00018 00026 00034 00042 00050 00058
 00003 00011 00019 00027 00035 00043 00051 00059
 00004 00012 00020 00028 00036 00044 00052 00060
 00005 00013 00021 00029 00037 00045 00053 00061
 00006 00014 00022 00030 00038 00046 00054 00062
 00007 00015 00023 00031 00039 00047 00055 00063
 00008 00016 00024 00032 00040 00048 00056 00064
File C:
 00204 00492 00780 01068 01356 01644 01932 02220
 00000 00000 00000 00000 00000 00000 00000 00000
 00000 00000 00000 00000 00000 00000 00000 00000
 00000 00000 00000 00000 00000 00000 00000 00000
 01356 03692 06028 08364 10700 13036 15372 17708
 00000 00000 00000 00000 00000 00000 00000 00000
 00000 00000 00000 00000 00000 00000 00000 00000
 00000 00000 00000 00000 00000 00000 00000 00000
 00000 00000 00000 00000 00000 00000 00000 00000
 00000 00000 00000 00000 00000 00000 00000 00000
 00000 00000 00000 00000 00000 00000 00000 00000
 ..... ..... ..... ..... ..... ..... ..... .....
 ..... ..... ..... ..... ..... ..... ..... .....
```

Strassen algorithm: a skeleton

```
/*
File name: out-of-core-strassen.c
/usr/local/mpi/bin/mpicc -0 out-of-core-strassen.c
*/

#include "sys/types.h"
#include "stdlib.h"
#include "stdio.h"
#include "string.h"
#include <argz.h>
#include <stdlib.h>
#include <limits.h>
#include <string.h>
#include <math.h>
#include <sys/time.h>
#include <sys/stat.h>
#include <unistd.h>
#include "mpi.h"

#define MPI_OFFSET_ZERO 0

#define P          4             /* number of tasks       */
#define Nb         8             /* slice size            */
#define Ng         ( Nb * P )    /* global size of matrix */

static char fileA[] = "fileA";
static char fileB[] = "fileB";
static char fileC[] = "fileC";

void
fill_file_A(char *FNAME, int N)
{
  int             i, data;
  FILE            *f;

  /* sprintf(s, FNAME, 0);*/
  if ((f = (FILE *) fopen (FNAME, "w+")) == NULL) {
    perror("io2: error in opening auxiliary files");
    exit(1);
  }
  for (i = 0; i < N * N; i++) {
    data = i + 1;
    if (fwrite(&data, sizeof(int), 1, f) != 1) {
      perror ("io3: can't write in an auxiliary file");
      fclose(f);
      exit(1);
    }
  }
  fclose(f);
}

void
fill_file_B(char *FNAME, int N)
{
  int             i, j, data;
  FILE            *f;

  /* sprintf(s, FNAME, 0);*/
  if ((f = (FILE *) fopen(FNAME, "w+")) == NULL) {
    perror("io2: error in opening auxiliary files");
    exit(1);
  }
```

```
    for (i = 0; i < N; i++) {
      for(j=0; j < N; j++) {
      data = (i+1) + N*j;
      if (fwrite(&data, sizeof(int), 1, f) != 1) {
        perror("io3: can't write in an auxiliary file");
        fclose(f);
        exit(1);
      }
      }
    }
    fclose(f);
  }

  /*
   * Print screen fname file (which much contain integers)
   */

  void
  aff(char *fname, int N)
  {
    FILE           *fp;
    int             i,count=0;

    fp = fopen(fname, "r");
    fread(&i, 4, 1, fp);
    while (0 == feof(fp)) {
      printf("%6.5d ", i);
      if(((count+1) % N) == 0) {
        count = 0; printf("\n");
      } else count++;
      fread(&i, 4, 1, fp);
    }
    printf("\n");
    fclose(fp);
  }

  void
  aff_in_core(int *t,int n){
    int i,j, count=0;

    for(i=0;i<n*n;i++) {
      printf("%6.5d",*(t+i));
      if(((count+1) % n) == 0) {
        printf("\n");count = 0;
      }  else  {
        count++;
      }
    }
  }

  /* c = a+b */
  void add_in_core(int n, int *a, int *b, int *c)
  {

    int i, j;

    for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
        *(c+(i*n)+j) = *(a+(i*n)+j) + *(b+(i*n)+j);
  }

  /* c = a-b */
  void sub_in_core(int n, int *a, int *b, int *c)
  {

    int i, j;

    for (i = 0; i < n; i++)
```

```
    for (j = 0; j < n; j++)
      *(c+(i*n)+j) = (*(a+(i*n)+j)) - (*(b+(i*n)+j));
  }

  /* c = a*b */
  void multiply_in_core(int n, int *a, int *b, int *c)
  {
    int i, j, k;

    for(i=0;i<n;i++)
      for(j=0;j<n;j++){
        *(c+(i*n)+j) = 0;
        for(k=0;k<n;k++)
          *(c+(i*n)+j) += *(a+(i*n)+k) * *(b+(k*n)+j);
      }
  }

  /* c = a*b (Strassen in-core Algorithm) */
  void multiply_strassen(int NN, int *a, int *b, int *c)
  {

    int i,j,k,n;
    int *a11, *a12, *a21, *a22;
    int *b11, *b12, *b21, *b22;
    int *M0, *M1, *M2, *M3, *M4, *M5, *M6;
    int *c1, *c2, *d1, *d2;
    int *c11, *c12, *c21, *c22;

    n = NN/2;

    /* initial step: isolate sub-matrices */

    a11 = (int *)malloc(n*n*sizeof(int));
    a12 = (int *)malloc(n*n*sizeof(int));
    a21 = (int *)malloc(n*n*sizeof(int));
    a22 = (int *)malloc(n*n*sizeof(int));
    b11 = (int *)malloc(n*n*sizeof(int));
    b12 = (int *)malloc(n*n*sizeof(int));
    b21 = (int *)malloc(n*n*sizeof(int));
    b22 = (int *)malloc(n*n*sizeof(int));

    for(i=0;i<n;i++)
      for(j=0;j<n;j++) {
        k = 2*(i*n) + j;
        *(a11 + (i*n) + j) = *(a + k);
        *(b11 + (i*n) + j) = *(b + k);
      }

    for(i=0;i<n;i++)
      for(j=0;j<n;j++){
        k = (i*n) + j + (n*(i+1));
        *(a12 + (i*n) + j) = *(a + k);
        *(b12 + (i*n) + j) = *(b + k);
      }

    for(i=0;i<n;i++)
      for(j=0;j<n;j++){
        k = 2*(i*n) + j + (n * Nb);
        *(a21 + (i*n) + j) = *(a + k);
        *(b21 + (i*n) + j) = *(b + k);
      }

    for(i=0;i<n;i++)
      for(j=0;j<n;j++){
        k = 2*(i*n) + j + (n*Nb) + n;
        *(a22 + (i*n) + j) = *(a + k);
        *(b22 + (i*n) + j) = *(b + k);
      }
```

```
    /*
      aff(a11,n);printf("\n");aff(a12,n);printf("\n");
      aff(a12,n);printf("\n");aff(a22,n);printf("\n");
    */
    /* first part */

    d1  = (int *)malloc(n*n*sizeof(int));
    d2  = (int *)malloc(n*n*sizeof(int));
    M0  = (int *)malloc(n*n*sizeof(int));
    M1  = (int *)malloc(n*n*sizeof(int));
    M2  = (int *)malloc(n*n*sizeof(int));
    M3  = (int *)malloc(n*n*sizeof(int));
    M4  = (int *)malloc(n*n*sizeof(int));
    M5  = (int *)malloc(n*n*sizeof(int));
    M6  = (int *)malloc(n*n*sizeof(int));
    c1  = (int *)malloc(n*n*sizeof(int));
    c2  = (int *)malloc(n*n*sizeof(int));
    c11 = (int *)malloc(n*n*sizeof(int));
    c12 = (int *)malloc(n*n*sizeof(int));
    c21 = (int *)malloc(n*n*sizeof(int));
    c22 = (int *)malloc(n*n*sizeof(int));

    add_in_core(n, a11, a22, d1);
    add_in_core(n, b11, b22, d2);
    multiply_in_core(n, d1, d2, M0);

    sub_in_core(n, a12, a22, d1);
    add_in_core(n, b21, b22, d2);
    multiply_in_core(n, d1, d2, M1);

    sub_in_core(n, b21, b11, d1);
    multiply_in_core(n, a22, d1, M2);

    add_in_core(n, a11, a12, d1);
    multiply_in_core(n, d1, b22, M3);

    add_in_core(n, a21, a22, d1);
    multiply_in_core(n, d1, b11, M4);

    sub_in_core(n, b12, b22, d1);
    multiply_in_core(n, a11, d1, M5);

    sub_in_core(n, a21, a11, d1);
    add_in_core(n, b11, b12, d2);
    multiply_in_core(n, d1, d2, M6);

    /* Second part */

    add_in_core(n, M0, M1, c1);
    sub_in_core(n, M2, M3, c2);
    add_in_core(n,c1, c2, c11);

    add_in_core(n, M3, M5, c12);

    add_in_core(n, M2, M4, c21);

    sub_in_core(n, M0, M4, c1);
    add_in_core(n, M5, M6, c2);
    add_in_core(n, c1, c2, c22);

    /* Third step: copy into the final destination */

    for(i=0;i<n;i++)
      for(j=0;j<n;j++) {
        k = 2*(i*n) + j;
        *(c + k) = *(c11 + (i*n) + j);
      }
```

```
    for(i=0;i<n;i++)
      for(j=0;j<n;j++){
        k = (i*n) + j + (n*(i+1));
        *(c + k) = *(c12 + (i*n) + j);
      }

    for(i=0;i<n;i++)
      for(j=0;j<n;j++){
        k = 2*(i*n) + j + (n * Nb);
        *(c + k) = *(c21 + (i*n) + j);
      }

    for(i=0;i<n;i++)
      for(j=0;j<n;j++){
        k = 2*(i*n) + j + (n*Nb) + n;
        *(c + k) = *(c22 + (i*n) + j);
      }
  }


  int
  main( int argc, char *argv[] )
  {
    int i, j, k;
    int iv;
    int myrank, commsize;
    int extent;
    int colext;
    int slicelen;
    int sliceext;
    int sizes[2], subsizes[2], starts[2];
    int length[ 3 ];
    MPI_Aint disp[ 3 ];
    MPI_Datatype type[ 3 ];
    MPI_Datatype ftypeA;
    MPI_Datatype ftypeB;
    MPI_Datatype ftypeC;
    MPI_Datatype slice_typeB;
    MPI_Datatype block_typeC;
    MPI_Datatype slice_typeC;
    int mode;
    MPI_File fh_A, fh_B, fh_C;
    MPI_Offset offsetA;
    MPI_Offset offsetB;
    MPI_Offset offsetC;
    MPI_Status status;
    int val_A[ Nb ] [ Ng ], val_B[ Ng ], [ Nb ], val_C[ Nb ] [ Nb ];

    /* initialize MPI */
    MPI_Init ( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
    MPI_Comm_size( MPI_COMM_WORLD, &commsize );

    /* We fill and we print the file contain */
    fill_file_A(fileA,Nb);
    fill_file_B(fileB,Nb);
    if(myrank == 0) {
      printf("File A:\n");
      aff(fileA, Nb);
      printf("File B:\n");
      aff(fileB,Nb);
    }
    /*  exit(0);*/

    /*****************************************/

    MPI_Type_extent( MPI_INT, &extent );
```

```
    colext = Nb * extent;
    slicelen = Ng * Nb;
    sliceext = slicelen * extent;

    /*
    The subarray type constructor creates an MPI datatype describing
    an n-dimensional subarray of an n-dimensional array. The subarray may
    be situated anywhere within the full array, and may be of any nonzero
    size up to the size of the larger array as long as it is confined
    within this array. This type constructor facilitates creating
    filetypes to access arrays distributed in blocks among processes to a
    single file that contains the global array.

    Note: { MPI_ORDER_C}
       The ordering used by C arrays, (i.e., row-major order)

    */
    /* create filetype for matrix A */
    sizes[0]=Nb; sizes[1]=Nb;
    subsizes[0]= Nb/2; subsizes[1]=Nb/2;
    starts[0]=(myrank % (P/2))*subsizes[0];
    starts[1]=((myrank / (P/2))*subsizes[1]);
    MPI_Type_create_subarray(2, sizes, subsizes, starts, MPI_ORDER_C,
                             MPI_INT, &ftypeA);

    MPI_Type_commit( &ftypeA );

    /* create filetype for matrix B */
    MPI_Type_create_subarray(2, sizes, subsizes, starts, MPI_ORDER_C,
                             MPI_INT, &ftypeB);
    MPI_Type_commit( &ftypeB );

    /* create filetype for matrix C */
    MPI_Type_vector( Nb, Nb, Ng, MPI_INT, &block_typeC );
    MPI_Type_hvector( P, 1, colext, block_typeC, &slice_typeC );
    length[ 0 ] = 1;
    length[ 1 ] = 1;
    length[ 2 ] = 1;
    disp[ 0 ] = 0;
    disp[ 1 ] = sliceext * myrank;
    disp[ 2 ] = sliceext * P;
    type[ 0 ] = MPI_LB;
    type[ 1 ] = slice_typeC;
    type[ 2 ] = MPI_UB;
    MPI_Type_struct( 3, length, disp, type, &ftypeC );
    MPI_Type_commit( &ftypeC );

    /* open files */
    mode = MPI_MODE_RDONLY;
    MPI_File_open( MPI_COMM_WORLD, fileA, mode, MPI_INFO_NULL, &fh_A );
    mode = MPI_MODE_RDONLY;
    MPI_File_open( MPI_COMM_WORLD, fileB, mode, MPI_INFO_NULL, &fh_B );
    mode = MPI_MODE_CREATE | MPI_MODE_WRONLY;
    MPI_File_open( MPI_COMM_WORLD, fileC, mode, MPI_INFO_NULL, &fh_C );

    /* set views */
    MPI_File_set_view (fh_A, MPI_OFFSET_ZERO, MPI_INT, ftypeA,
                       "native", MPI_INFO_NULL );
    MPI_File_set_view( fh_B, MPI_OFFSET_ZERO, MPI_INT, ftypeB,
                       "native", MPI_INFO_NULL );
    MPI_File_set_view( fh_C, MPI_OFFSET_ZERO, MPI_INT, ftypeC,
                       "native", MPI_INFO_NULL );

    offsetA = MPI_OFFSET_ZERO;
    offsetB = MPI_OFFSET_ZERO;
    offsetC = MPI_OFFSET_ZERO;

    /* read a block of A */
```

```
    MPI_File_read_at_all( fh_A, offsetA, val_A, (Nb/2)*(Nb/2),
                          MPI_INT, &status );
    printf("Block of A (PID=%d)\n" ,myrank);
    aff_in_core(val_A,Nb/2);

    /* read a block of B */

    MPI_File_read_at_all( fh_B, offsetB, val_B, (Nb/2)*(Nb/2),
                          MPI_INT, &status );
    printf("Block of B (%d)\n" ,myrank);
    aff_in_core(val_B,Nb/2);

    /* close files */
    MPI_File_close( &fh_A );
    MPI_File_close( &fh_B );
    MPI_File_close( &fh_C );

    /* free filetypes */
    MPI_Type_free( &ftypeA );
    MPI_Type_free( &ftypeB );
    MPI_Type_free( &ftypeC );

    if (myrank == 0) {
      printf ("File C:\n");
      /* aff(fileC,Nb); */
    }

    /* finalize MPI */
    MPI_Finalize();
  }
  /****************************
   * Strassen in-core algorithm
   ****************************
  int
  main(){

    int *A,*B,*C;
    int i,j,k;

    A = (int *)malloc(Nb*Nb*sizeof(int));
    B = (int *)malloc(Nb*Nb*sizeof(int));
    C = (int *)malloc(Nb*Nb*sizeof(int));

    fill_A(A,Nb);
    printf("Matrix A:\n");
    aff(A_Nb);

    fill_B(B,Nb);
    printf("Matrix B:\n");
    aff(B,Nb);

    multiply_strassen(Nb,A,B,C);

    printf("Matrix C:\n");
    aff(C,Nb);
  }

  ************************/


  /******* Execution Trace
  [cerin@fatboy cerin]$ /usr/local/mpi/bin/mpicc -0 out-of-core-strassen.c
  [cerin@fatboy cerin]$ /usr/local/mpi/bin/mpirun -np 4 a.out
  File A:
   00001 00002 00003 00004 00005 00006 00007 00008
   00009 00010 00011 00012 00013 00014 00015 00016
   00017 00018 00019 00020 00021 00022 00023 00024
   00025 00026 00027 00028 00029 00030 00031 00032
```

```
00033 00034 00035 00036 00037 00038 00039 00040
00041 00042 00043 00044 00045 00046 00047 00048
00049 00050 00051 00052 00053 00054 00055 00056
00057 00058 00059 00060 00061 00062 00063 00064

File B:
00001 00009 00017 00025 00033 00041 00049 00057
00002 00010 00018 00026 00034 00042 00050 00058
00003 00011 00019 00027 00035 00043 00051 00059
00004 00012 00020 00028 00036 00044 00052 00060
00005 00013 00021 00029 00037 00045 00053 00061
00006 00014 00022 00030 00038 00046 00054 00062
00007 00015 00023 00031 00039 00047 00055 00063
00008 00016 00024 00032 00040 00048 00056 00064

Block of A (PID=0)
00001 00002 00003 00004
00009 00010 00011 00012
00017 00018 00019 00020
00025 00026 00027 00028
Block of B (PID=0)
00001 00009 00017 00025
00002 00010 00018 00026
00003 00011 00019 00027
00004 00012 00020 00028
File C:
Block of A (PID=3)
00037 00038 00039 00040
00045 00046 00047 00048
00053 00054 00055 00056
00061 00062 00063 00064
Block of B (PID=3)
00037 00045 00053 00061
00038 00046 00054 00062
00039 00047 00055 00063
00040 00048 00056 00064
Block of A (PID=1)
00033 00034 00035 00036
00041 00042 00043 00044
00049 00050 00051 00052
00057 00058 00059 00060
Block of B (PID=1)
00005 00013 00021 00029
00006 00014 00022 00030
00007 00015 00023 00031
00008 00016 00024 00032
Block of A (PID=2)
00005 00006 00007 00008
00013 00014 00015 00016
00021 00022 00023 00024
0029 00030 00031 00032
Block of B (PID=2)
00033 00041 00049 00057
00034 00042 00050 00058
00035 00043 00051 00059
00036 00044 00052 00060

[cerin@fatboy cerin]$

****************************/
```

Appendix 2: Selected Web Sites Related to I/O

Christophe Cérin, Université de Picardie Jules Verne, LaRIA, 5 rue du moulin neuf, 80000 Amiens, France
Hai Jin, Huazhong University of Science and Technology, Wuhan, 430074, China

Overview

http://www.buyya.com/superstorage An introduction and samples of the book "High Performance Mass Storage and Parallel I/O" (Hai Jin, Toni Cortes, Rajulmar Buyya) edited by IEEE and WILEY.
See also IEEE Press. (http://www.ieee.org/press);

http://www.ieeeftcc.org Web site of IEEE Task Force on Cluster Computing; See also http://www.clustercomputing.org/ for the newsletter;

http://www.cs.dartmouth.edu/pario/ Parallel I/O Archive at Dartmouth college;

http://www.msstc.org/ Mass Storage Systems Technical Committee web site;

http://www.laria.u-picardie.fr/~cerin/=paladin/ Parallel algorithms for sorting on heterogeneous clusters;

http://www.snia.org Storage terminology;

http://www.techfest.com/hardware/storage.htm General Storage Information:

http://www.unet.univie.ac.at/~a9405327/glossary/glossary_html.html Glossary on Parallel I/O;

http://www.raid-advisory.com Raid Advisory Board;

http://www.minet.net/linux/HOWTO-fr/Software-RAID-HOWTO The software RAID HOW-TO;

http://www.viarch.org VIA (Virtual Architecture Interface) technology;

http://www.infiniband.org InfiniBand technology;

http://sourceforge.net/foundry/storage Distributed file system repository: CODA, XFS, GFS...;

http://www.gfdl.gov/~vb/mpp_io.html mpp_io is a set of simple calls for parallel I/O on distributed systems. It is geared toward the writing of data in netCDF format;

http://hdf.ncsa.uiuc.edu/HDF5/ HDF5 is a general purpose library and file format for storing scientific dat;

http://www.unidata.ucar.edu/packages/netcdf/: NetCDF (Network Common Data Form) is an interface for array-oriented data access and a library that provides an implementation of the interface;

http://www.p2pwg.org/ Peer-to-peer working group. You will also find "white papers" on file systems; see also Jxta (http://www.jxta.org) and HailStorm languages (http://www.microsoft.com/net/hailstorm.asp);

http://www.acm.org/sigkdd ACM Special Interest Group on Knowledge Discovery in Data and Data Mining;

http://www.research.microsoft.com/research/datamine/acm-contents.htm Comm. ACM Special Issue on Data Mining (Vol. 39, No. 11, Nov. 1996);

http://www.cs.bham.ac.uk/~anp/TheDataMine.html The Data Mine;

http://www.isi.edu/nsf NSF Data Mining Workshop;

http://www.research.microsoft.com/datamine Data Mining and Knowledge Discovery Journal;

http://www.informatik.uni-trier.de/~ley/db/groups.html Database Research Groups;

http://www.almaden.ibm.com/cs/quest Quest Data Mining Home Page;

http://lib.stat.cmu.edu/~bill/DMLIST.html URLs for Data Mining;

http://www.research.microsoft.com/~fayyad/advances-kdd/fayap.html Usama Fayyad et al., Advances in Knowledge Discovery and Data Mining, AAAI Press, Menlo Park, Calif., 1996; MIT Press, Cambridge, Mass., 1996;

http://www-users.cs.umn.edu/~mjoshi/hpdmtut/index.htm V. Kumar and M. Joshi, Tutorial on High Performance Data Mining, Dept. of Computer Science, Univ. of Minnesota, 1999;

http://www.bell-labs.com/project/serendip/talks/tutorial.ps.gz K. Rastogi and K. Shim, "Tutorial on Scalable Algorithms for Mining Large Databases," Proc. Fifth ACM SIGKDD Int'l Conf. Knowledge Discovery & Data Mining, ACM Press, New York, 1999;

http://www-db.stanford.edu/~ullman/mining/mining.html Jeffrey D.Ullman notes on Data Mining;

http://www.acm.org/sigmod/databaseSoftware/ open source databases softwares;

http://gist.cs.berkeley.edu/ The GiST Indexing Project. The GiST project studies the engineering and mathematics behind content-based indexing for massive amounts of complex content. The project consists of a number of components: The basis of our work is the Generalized Search Tree (GiST), a template indexing structure that allows domain experts (e.g in computer vision, bioinformatics, or remote sensing) to easily customize a database system to index their content.

Benchmarking

http://www.textuality.com/bonnie/: testing the file system level and disks;

http://www.acnc.com/benchmarks.html: this site is a MUST because you find almost everything about IO benchmarking on Unix or Windows platforms;

ftp://samba.org/pub/tridge/dbench (Dbench);

http://www.netapp.com/tech_library/3022.html (postmark);

http://www.coker.com.au/bonnie++ (Bonnie++);

http://www.netapp.com/tech_library/3022.html (PostMark: A New File System Benchmark);

http://www.acnc.com/04_02.html: web site with many benchmarks;

http://www.namesys.com/benchmarks/mongo/mongo_readme.html (Mongo);

http://etestinglabs.com/benchmarks/netbench/netbench.asp (NetBench is a portable benchmark program that measures how well a file server handles file I/O requests from 32-bit Windows clients). See also

http://etestinglabs.com/ (Etestinglabs) or http://www.zdnet.com/ (ZDnet);

ftp://ftp.arl.mil/pub/ttcp/ttcp.dpk/ testing the network layer.

Index

A-B

Index

C

Index

D

Index

E-L

Index

M-O

Index

P-R

Index

S

Index

T-Z

List of Figures

Chapter 1: Motivating I/O Problems and their Solutions

Chapter 2: Parallel Sorting on Heterogeneous Clusters

Chapter 3: A Sensitivity Study of Parallel I/O under PVFS-Lessons Learned Using a Parallel File System on PC Clusters

Chapter 4: The Parallel Effective I/O Bandwidth Benchmark-b_eff_io

Chapter 5: Parallel Join Algorithms on Clusters

Chapter 6: Server-side Scheduling in Cluster Parallel I/O Systems

Chapter 7: Design and Implementation of a Large Virtual NVRAM Cache for Software RAID

List of Tables

Chapter 2: Parallel Sorting on Heterogeneous Clusters

Chapter 4: The Parallel Effective I/O Bandwidth Benchmark-b_eff_io

Chapter 5: Parallel Join Algorithms on Clusters

Chapter 6: Server-side Scheduling in Cluster Parallel I/O Systems

Chapter 7: Design and Implementation of a Large Virtual NVRAM Cache for Software RAID

List of Examples

Chapter 1: Motivating I/O Problems and their Solutions

Chapter 2: Parallel Sorting on Heterogeneous Clusters