HIGH PERFORMANCE COMPUTING SYSTEMS AND APPLICATIONS

edited by
Jonathan Schäffer

KLUWER ACADEMIC PUBLISHERS
HIGH PERFORMANCE COMPUTING SYSTEMS AND APPLICATIONS

edited by

Jonathan Schaeffer
University of Alberta, Canada

KLUWER ACADEMIC PUBLISHERS
Boston / Dordrecht / London
Contents

Preface xi

Part I Keynote Presentations

1 Making Computer Design a Science Instead of an Art
   John L. Gustafson 3

2 Scalable Shared Memory and Shared Memory Clustering
   Marc Snir 15

3 High-Performance Computational Grids
   Ian Foster 17

4 High Performance Computing and Computational Chemistry in Canada—Quo Vadis
   Dennis Salahub 19

5 Scalable Shared-Memory Multiprocessing: The Future of High-Performance Computing
   Daniel Lenoski 21

6 Challenges of Future High-End Computing
   David H. Bailey 23
Part II  Computational Physics

7  Parallel Performance of a Code Using Multidimensional FFTs
   (Invited Paper)
   John A. Zollweg  37

8  Object-Oriented Scientific Programming with Fortran 90 (Invited Paper)
   Charles D. Norton  47

9  Alternating Direction Implicit Methods on Distributed and Shared
    Memory Parallel Computers (Invited Paper)
   Robert Rankin and Igor Voronkov  59

10 Simulating Galaxy Formation on SMPs
    Robert Thacker, Hugh Couchman and Frazer Pearce  71

Part III  Database Systems

11 Data Parallelism in Java
   Michael Philippson  85

12 A Comparative Study of Soft Real-Time Transaction Scheduling
    Policies in Parallel Database Systems
   Sonia Takkar and Sivarama P. Dandamudi  101

13 An Age-Threshold Algorithm for Garbage Collection in Log-Structured
    Arrays and File Systems
   Jai Menon and Larry Stockmeyer  119

14 Thinking Big in a Small World — Efficient Query Execution on
    Small-scale SMPs
   Stefan Manegold and Florian Waas  133

Part IV  Parallel I/O Issues

15 Speeding Up Automatic Parallel I/O Performance Optimization
    in Panda
   Ying Chen and Marianne Winslett  149
<table>
<thead>
<tr>
<th>Page</th>
<th>Title</th>
<th>Author(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
<td>Software Caching in A Parallel I/O Runtime System to Support</td>
<td>Sung-Soon Park, Jaechun No, Jesús Carretero and Alok Choudhary</td>
</tr>
<tr>
<td>47</td>
<td>HIPPI-6400—Designing for Speed</td>
<td>Don E. Tolmie</td>
</tr>
<tr>
<td>59</td>
<td>RSD — Resource and Service Description</td>
<td>Matthias Brune, Jörn Gehring, Axel Keller and Alexander Reinefeld</td>
</tr>
<tr>
<td>71</td>
<td>An Examination of the Performance of Two Electromagnetic Simulations</td>
<td>Daniel S. Katz, Tom Cwik and Thomas Sterling</td>
</tr>
<tr>
<td></td>
<td>on a Beowulf-Class Computer</td>
<td></td>
</tr>
<tr>
<td>85</td>
<td>Part VI Computational Chemistry</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>NWChem Molecular Dynamics Simulation (Invited Paper)</td>
<td>T.P. Straatsma</td>
</tr>
<tr>
<td>119</td>
<td>Using Gaussian Geminal Functions in Molecular Calculations: Integral</td>
<td>Andrew Komornicki</td>
</tr>
<tr>
<td></td>
<td>Evaluation (Invited Presentation)</td>
<td></td>
</tr>
<tr>
<td>133</td>
<td>The Development of Parallel GAMESS (Invited Presentation)</td>
<td>Michael W. Schmidt</td>
</tr>
<tr>
<td>149</td>
<td>Applications/Technology Push/Pull: Building Computational Infrastructure</td>
<td>Peter Taylor</td>
</tr>
<tr>
<td></td>
<td>(Invited Presentation)</td>
<td></td>
</tr>
<tr>
<td>163</td>
<td>Part VII NUMA Architectures</td>
<td></td>
</tr>
<tr>
<td>177</td>
<td>A Virtual Cache Scheme for Improving Cache-Affinity on</td>
<td>Dongwook Kim, Eunjin Kim and Joonwon Lee</td>
</tr>
<tr>
<td></td>
<td>Multiprogrammed Shared Memory Multiprocessors</td>
<td></td>
</tr>
</tbody>
</table>

Contents vii
HIGH PERFORMANCE COMPUTING SYSTEMS

26
Stable Performance for cc-NUMA using First-Touch Page Placement and Reactive Proxies
Sarah A. M. Talbot and Paul H. J. Kelly

27
A Comparative Study of Cache-Coherent Nonuniform Memory Access Systems
Gheith A. Abandah and Edward S. Davidson

28
Parallel Computation of the Diameter of a Graph
Jean-Albert Ferrez, Komei Fukuda and Thomas M. Liebling

Part VIII Computational Biology

29
Calculation of Solvation Free Energies
Enrico O. Purisima

30
Molecular Docking With a View: The Integration of a Monte Carlo Docking Program into a Virtual Reality Environment
Trevor Hart, Richard Gillilan, Ryan Lilien, Steven Ness and Randy Read

31
Structural Ribonucleic Acid Motifs Identification And Classification
Patrick Gendron, Daniel Gautheret and Francois Major

32
Computational Protein Folding
John R. Gunn

Part IX Applications

33
Distributed Computing Over High-Speed Local Area Networks
Sherali Zeadally

34
Fast and Efficient Parallel Matrix Computations on a Linear Array with a Reconfigurable Pipelined Optical Bus System
Kegin Li, Yi Pan and Si-Qing Zheng

35
Development of Parallel, Adaptive Finite Element Implementations for Modelling Two-Phase Oil/Water Flow
Donald J. Morton
<table>
<thead>
<tr>
<th>Page</th>
<th>Title</th>
<th>Authors</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>251</td>
<td>Parallel Solution Alternatives for Sparse Triangular Systems in Interior Point Methods</td>
<td>Huseyin Simitci</td>
<td>391</td>
</tr>
<tr>
<td>267</td>
<td>Part X General Session</td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>Managing Long Linked Lists Using Lock-Free Techniques</td>
<td>Mohammad Farook and Peter Graham</td>
<td>407</td>
</tr>
<tr>
<td>283</td>
<td>Single Stepping in Concurrent Executions</td>
<td>Thomas Kunz and Marc Khousam</td>
<td>423</td>
</tr>
<tr>
<td>299</td>
<td>Adaptive Parallelism on a Network of Workstations</td>
<td>Mohan V. Nibhanupudi and Boleslaw K. Szymanski</td>
<td>439</td>
</tr>
<tr>
<td>309</td>
<td>A New Family of Ring-Bisection Networks and Routing Algorithms</td>
<td>Yuchong Sun, Paul Y.S. Cheung and Xiaola Lin</td>
<td>453</td>
</tr>
<tr>
<td>323</td>
<td>Index</td>
<td></td>
<td>463</td>
</tr>
</tbody>
</table>
This book contains the proceedings of HPCS'98, the 12th annual international symposium on High Performance Computing Systems and Applications. The HPCS series originally started in 1987 as a two-day conference (Supercomputing Symposium '87) at the University of Calgary. By bringing Canadian researchers with a common interest in supercomputing together, the intent was to create an academic, industrial and government environment for fostering the growth of computational research. In 1991 the symposium broadened its scope to include international papers, and changed its name to High Performance Computing Systems (HPCS).

There are a plethora of conferences that advertise using the keywords parallel, distributed and/or high-performance computing. With so much competition, it is imperative that an event be attractive to prospective authors and attendees. Towards this goal, several things were tried to strengthen the conference:

1. Changing the program emphasis to high performance computing and applications. This will allow the HPC system developers and the end users to get together and interact.

2. Offering an impressive lineup of international high-profile keynote and invited speakers.

3. Increasing the refereeing standards. Only 38% of submitted papers were accepted.


5. Adding a poster session, allowing students to showcase their work.

6. Locating the conference at a unique venue. I doubt many of the attendees will soon forget the novelty of West Edmonton Mall.

Feedback suggests that all these changes enhanced the conference. It is hoped that next year's conference can build on our success.
This book contains the keynote, invited and refereed papers of HPCS’98. For the keynote speakers, submitting a paper was optional. I would like to express my sincere thanks to John Gustafson and David Bailey for kindly turning their excellent presentations into papers for this book. The annual C3 meeting was help at HPCS’98; Brian Unger (University of Calgary) and Sid Karin (San Diego Supercomputer Center) gave overviews on the current state of HPC in Canada and the United States. Phil Tannenbaum (NEC) gave a light-hearted talk on the correlation of computer price, performance and weight. Sadly, Phil did not have time to turn his talk into a paper for this book.

HPCS’98 gratefully acknowledges the sponsorship of the following companies (in alphabetical order): C3, Canarie, Digital Equipment Corporation, HNSX/NEC, IBM, IEEE Canada, Silicon Graphics, Sun Microsystems, Super-Can, and Telus. The University of Alberta provided support, as well as the Faculty of Science, Computing and Networking Services, and the Department of Chemistry, Computing Science and Physics.

I would also like to thank the program committee, which included: Hamid Arabnia (University of Georgia, U.S.A.), Henri Bai (Vrije Universiteit, The Netherlands), Luc Bauwens (University of Calgary, Canada), Virendra Bhavsar (University of New Brunswick, Canada), Brian d’Auriol (University of Akron, U.S.A.), Geoffrey Fox (Syracuse University, U.S.A.), Guang Gao (University of Delaware, U.S.A.), Peter Graham (University of Manitoba, Canada), Qian Ping Gu (University of Aizu, Japan), Laurie Hendren (McGill University, Canada), Robert Ito (University of British Columbia, Canada), Wayne Karpoff (Myrias Computer Technologies Corp., Canada), Mariusz Klobukowski (University of Alberta, Canada), Thomas Kunz (Carleton University, Canada), Christian Lengauer (University of Passau, Germany), Keqin Li (SUNY, New Paltz, U.S.A.), Michael Quinn (Oregon State University, U.S.A.), Alexander Reinefeld (Paderborn Center for Parallel Computing, Germany), John Samson (University of Alberta, Canada), Ken Sevecik (University of Toronto, Canada), Ajit Singh (University of Waterloo, Canada), Rick Sydora (University of Alberta, Canada), Duane Szafron (University of Alberta, Canada), Ron Unrau (University of Alberta, Canada), and David Wishart (University of Alberta, Canada).

Organizing a large conference like this is not possible without the help of many people. In particular, I would like to thank: Sylvia Boyetchko, Michael Byrne, George Carmichael, Sunrose Ko, Marc LaFrance, Nadine Leenders, Wally Lysz, Steve MacDonald, Diego Novillo, Janet Service, Ron Senda, John Samson, Denise Thornton, Ron Unrau, and Norris Weimer. Steve MacDonald and Diego Novillo need to be singled out for their enormous commitment to the conference. I would especially like to thank the organizers of the applications streams: Mariusz Klobukowski (chemistry), Rick Sydora (physics) and David Wishart (biology).

Finally, we appreciate the support of Scott Delman from Kluwer Academic Press for giving us the opportunity to publish these proceedings.

JONATHAN SCHAEFFER
jonathan@cs.ualberta.ca
SOFTWARE CACHING IN A PARALLEL I/O RUNTIME SYSTEM TO SUPPORT IRREGULAR APPLICATIONS

Sung-Soon Park¹, Jaechun No², Jesús Carretero³
and Alok Choudhary⁴

¹ Department of Computer Science and Engineering, Anyang University, Korea
² Department of Engineering and Computer Science, Syracuse University, U.S.A.
³ Arquitectura y Tecnología de S. I., Universidad Politécnica de Madrid, Spain
⁴ Department of Electrical and Computer Engineering, Northwestern University, U.S.A.

sspark@ece.nwu.edu
jno@ece.nwu.edu
jcarrere@ece.nwu.edu
choudhar@ece.nwu.edu

Abstract: In this paper we present the design and implementation of a runtime system based on collective I/O techniques for irregular applications. The design is motivated by the requirements of a large number of science and engineering applications, including teraflops applications, where the data is required to be reorganized into a canonical form for further processing or restarting. We also propose the design and implementation of a software caching method to improve the performance of the collective I/O techniques. The main idea of the software caching method is that, in an irregular application, the same data may be accessed repeatedly during the execution of subsequent irregular loops. Thus, improved performance can be achieved by storing the data in a processor's local memory and using it in the subsequent loops. We show the performance results for two implementations: non-caching and software caching methods. The performance results were collected on an Intel Paragon machine located at Caltech.

Keywords: software caching, parallel I/O, irregular applications, collective I/O.
16.1 INTRODUCTION

In parallel computing environments, I/O intensive applications have been increasing since many of the large-scale applications are data intensive rather than compute intensive (Poole, 1994; Rosario and Choudhary, 1994). These applications have very large I/O needs, and are further complicated by having different I/O types (such as checkpointing of large data sets for restarting, or periodically writing snapshots of the computation for subsequent visualization). A large number of these applications are irregular, where accesses to data are performed through one or more levels of indirection (Carretero et al., 1998; No et al., 1998). Sparse matrix computations, particle codes, and many CFD applications, where geometries and meshes are described via indirections, exhibit this characteristic. In I/O intensive applications, especially for irregular problems, the I/O cost for reading and writing data is more significant than communication or computational overheads. To reduce the I/O cost and to enhance performance, collective I/O has been used as a general mechanism (Choudhary et al., 1994). In collective I/O, processors cooperate to combine several fine-grained I/O requests into a single, canonically ordered, coarse-grained request.

In previous works (Carretero et al., 1998; No et al., 1998), we proposed some preliminary ideas to provide parallel I/O for irregular applications. In this paper, we present the design and implementation of a software caching method, using collective I/O, to produce high performance I/O. The motivation is that, in irregular applications, the same data may be accessed repeatedly during an execution. Therefore, some I/O operations can be eliminated by storing the reused data in the processor's local memory. The experimental results, obtained on the Intel Paragon at Caltech, show the feasibility of the software caching method proposed in this paper.

The rest of the paper is organized as follows. In Section 16.2, we present an overview of irregular applications and our collective I/O method to support them. Section 16.3 presents the software caching method. Experimental results for the non-caching and software caching methods are shown in Section 16.4. Finally, some conclusions are presented in Section 16.5.

16.2 I/O FOR IRREGULAR APPLICATION

In irregular applications, the data domain area to be accessed is determined by computing one or more level of indirections. Figure 16.1 illustrates an irregular loop (Das et al., 1994). Arrays x and y are data arrays and a(ij), b(ij), c(j) and d(j), which are used to reference data, are called indirection arrays. This irregular problem can be abstracted into the form shown in Figure 16.2. In this abstraction, we assume that data is distributed using some partitioning scheme, which may be application dependent. There is an indirection array on each node that describes the location of the corresponding data elements in a global array. For example, in Figure 16.2, processor 0 accesses x(a(0)), x(a(4)), x(a(5)), x(a(2)), and processor 1 accesses x(a(7)), x(a(8)), x(a(6)), x(a(9)), etc.

The objective of the I/O library is to provide a high-level interface for the programmer to write code for reading/writing data from/to files, in the order imposed by the global array. There are several characteristics to be considered in developing an
I/O library for supporting irregular problems. One of them is that irregular problems often generate fine-grained data distributions requiring access to non-contiguous locations in a global array. Therefore, an appropriate collective I/O method is necessary to obtain high I/O performance.

16.2.1 Collective I/O

The design of the collective I/O library relies on the two-phase I/O strategy (Choudhary et al., 1994). The basic idea behind the two-phase collective I/O is to optimize at runtime the access patterns seen by the I/O system. In other words, a large number of small and disjoint I/O requests are converted into a small number of large contiguous requests. This optimization incurs costs in terms of additional communication and buffer space requirements. However, since communication speed is normally several orders of magnitude faster than the I/O speed, the overhead is smaller than the reduction in the I/O cost. Several factors must be considered in the design of a library based on this technique: buffer size used by the library, communication schedule con-
struction and reorganization, the number of processors participating in I/O at any time, and scheduling of I/O requests. In addition, for irregular computations, the number of passes through the data sets for computing an I/O schedule and reorganizing data are also important.

The collective read operations involve three basic steps: schedule construction, reading data from files, and redistributing data into the appropriate locations on each processor. In the write operations, a redistribution step precedes the file write step. Each of these steps consists of several phases, which are described below.

### 16.2.2 Schedule Construction

The schedule describes the communication and I/O patterns required for each node participating in an I/O operation. For regular multidimensional arrays, the accesses can be described using regular section descriptors. The communication schedule can be built based upon this information. On the other hand, when indirection arrays are involved in referencing the data array, they must be scanned to consider each element individually to determine its place in the global canonical representation as well as its destination processor.

Two factors affect the schedule construction in particular, and the overall I/O library design and performance in general. They are:

1. **Chunk Size**, which is the amount of buffer space available to the runtime library for I/O operations. For example, if the total size of the data per processor to be read/written is 8 MB and the chunk size is 2 MB, then the I/O operation will require four iterations to complete. A schedule must be built for each of these iterations.

2. **Number of processors involved in I/O**, which determines the communication among processors for data redistribution.

The following briefly describes the steps involved in schedule computation:

- Based upon the chunk size, each processor is assigned a data domain for which it is responsible for reading or writing. For example, if there are four processors and 16 elements to be read/written, with a buffer space of two elements on each processor, the chunk size is 8 elements. Processor 0 will be responsible for elements 0, 1, 8, and 9, processor 1 will be responsible for elements 2, 3, 10, and 11, and so on. For a chunk, each processor computes its part of data to read or write while balancing I/O workload. Next, with each index value in its local memory, each processor first decides from which chunk the appropriate data must be accessed, and then determines which processor is responsible for manipulating the data chunk.

- Index values in the local memory are rearranged into the reordered-indirection array based on the order of the destination processors. Therefore, we can communicate consecutive elements between processors (communication coalescing.)

Note that the irregular steps, and

### 16.2.3 A processor or writer

A processor or writer of process

then each processor collects

Let $C$ be the collective

Let $C$ be the collective

Let $C$ be the collective

Since in the area can

New sections

New sections

New sections

New sections

New sections

New sections
time, or of
figure, each
step.

de
uses
can
visit
ary
ary
will
on
on
other
other
or
or

step 1 \[ \forall i, 0 \leq i \leq C-1 \]
step 2 \[ \forall j, 0 \leq j \leq P-1 \]
\[ p_d = \text{compute} \_\text{destination} \_\text{processor}(p_j), 0 \leq d \leq P-1 \]
\[ \text{send}(p_j, p_d, S_{\text{index}}(i, p_d)) \]
\[ \text{recv}(p_j, p_d, R_{\text{index}}(i, p_d)) \]
\[ \text{send}(p_j, p_d, S_{\text{data}}(i, p_d)) \]
\[ \text{recv}(p_j, p_d, R_{\text{data}}(i, p_d)) \]
step 3 \[ \text{write}(\text{data} \_\text{file}, c_i, \text{offset}(i)), 0 \leq \text{mynode} \leq P-1 \]

Figure 16.3 The collective write operation algorithm.

Note that once it is constructed, the schedule information can be used repeatedly in
the irregular problems, whose access pattern is not changed during the computation
steps, and thereby amortizing its cost.

16.2.3 Parallel Collective Read/Write Operations

A processor involved in the computation is also responsible for reading data from files
or writing data into files. Let \( D \) bytes be the total size of data and \( P \) be the number
of processors. If the size of the data chunk is the same as the total size of the data,
then each processor reads \( D/P \) bytes of data from the file and distributes it among
processors based on the schedule information. For the case of writing, each processor
collects \( D/P \) bytes of data from other processors, and then writes it to the file. By
performing I/O this way, the workload can be evenly balanced across processors.

Let \( C \) be the number of data chunks. Let \( S_{\text{index}}(i, p_j) \) and \( S_{\text{data}}(i, p_j) \) be the size
of the index and the data to be sent to \( p_j \) in data chunk \( i \), respectively. Let \( R_{\text{index}}(i, p_j) \)
and \( R_{\text{data}}(i, p_j) \) be the size of the index and the data to be received from \( p_j \) in data
chunk \( i \), respectively. Then, \( \text{send}(o, d, \text{data}) \) sends data from processor \( o \) to \( d \),
\( \text{recv}(d, o, \text{data}) \) receives data from processor \( o \) to \( d \), and \( \text{write}(\text{fid}, \text{data}, \text{offset}) \) writes data in position offset of file fid. The steps involved in the
collective write operations are shown in Figure 16.3.

16.3 SOFTWARE CACHING

Since irregular applications spend too much time in accessing target data, if a data
area can be maintained and reused (Das et al., 1993), we can reduce the I/O cost.
Figure 16.4 shows an example of data reuse, where \( a(i) \) and \( c(j) \) are index values used
to reference data in the global array \( x \), with data overlapping between the two loops.
Re-referenced data is data which has been accessed in a prior loop and that will be
accessed in a subsequent loop. So, it will not be read from the file again when the
subsequent loop is executed. Released data has been accessed in a prior loop and will
not be accessed in a subsequent loop. Since this data will no longer be referenced,
it should be written back to the file when the execution of the prior loop is finished.
New data is defined as the data referenced in a loop that has never been accessed in
prior loops. Therefore, it is necessary to read this data from the file before executing
subsequent loops.
In this section, we propose a software caching method that exploits the data reuse property, and also show its implementation details.

16.3.1 Software Caching Method

The motivation for the software caching method is that, in irregular applications, the same data may be accessed repeatedly during the execution of irregular loops. The basic goals for the software caching method are as follows. First and foremost, it is to reduce I/O to the maximum extent possible. To achieve this, an I/O phase, read or write, is divided into two new I/O phases, where the second phase only accesses new data. The second goal is to reuse the schedule information constructed in the beginning, and to build only incremental schedule information for new data.

To satisfy these aims, we added the following two steps to basic collective read operation: reading data partially from files and redistributing it into appropriate locations of each processor, and performing s/w caching phase to modify schedule information. In the write operation, redistribution step precedes the file write step.

16.3.2 Applying the Software Caching Method

Figure 16.5 represents the execution pattern of the irregular problem in Figure 16.1, with and without software caching. The communication and I/O patterns required for each processor are determined in the schedule phase. I/O phases to execute irregular loops are inserted before and after each loop. In Figure 16.5, the read1 phase reads data from a file, and write1 phase writes data to the file after finishing computations, where \( x(a(ub1)) \sim x(a(ub1)) \) or \( x(c(lb2)) \sim x(c(lb2)) \) represent data to be accessed in these phases. In this example, the re-referenced data need not be written back to the file in the first write1 phase, because it will be used again in the read1 phase.

Figure 16.5(b) illustrates the execution phases when using the software caching method. read1 and write1 are almost the same as for the I/O operations without

---

In the diagram:

- **PO**: Represents the initial state where data is loaded.
- **PI**: Indicates a point where data is accessed.
- **P2**: Represents a point where data is updated or written back.
- **Lpg1**: Shows a referencing region by index of data.
- **UbG1**: Indicates a user group with data.
- **UbG2**: Represents another user group with data.

Legend:
- **Filled circle**: Released data
- **Double-lined square**: Re-referenced data
- **Empty rectangle**: New data

Figure 16.4 Example of data reuse.
(a) Non-caching Program

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Compute schedule information for loop1 and loop2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read1</td>
<td>Read all data which will be referred in loop1</td>
</tr>
<tr>
<td>Loop1</td>
<td>Write all data which has been referred in loop1</td>
</tr>
<tr>
<td>Read2</td>
<td>Read all data which will be referred in loop2</td>
</tr>
<tr>
<td>Write1</td>
<td>Write all data which has been referred in loop2</td>
</tr>
</tbody>
</table>

(b) SW Caching Program

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Compute schedule information for loop1 and loop2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read1</td>
<td>Read all data which will be referred in loop1</td>
</tr>
<tr>
<td>Loop1</td>
<td>Write all data which has been referred in loop1</td>
</tr>
<tr>
<td>Write2</td>
<td>Write all data which has been referred in loop2 but will not be referred in loop2</td>
</tr>
<tr>
<td>SW caching</td>
<td>Modify schedule information to access new data</td>
</tr>
<tr>
<td>Read2</td>
<td>Read all data which will be referred in loop2 but has never been referred in loop1</td>
</tr>
<tr>
<td>Loop2</td>
<td>Write all data which has been referred in loop2</td>
</tr>
</tbody>
</table>

**Figure 16.5** Execution format for non-caching and SW caching programs.

Software caching. Read2 and Write2 are the phases for partially reading and writing data, based on the overlap area. In the Write2 phase, some data which will not be used in a subsequent loop is written to the file. Similarly, Read2 reads some data which has never been read from the file, but that will be referenced in a subsequent loop. Since only some part of the data is written/read to/from the file, the I/O cost for the Write2 and Read2 is expected to be smaller than for Write1 and Read1. In the SW caching phase, the schedule information for the new loop is reconstructed. It includes the number of re-referenced data chunks and the modified data domain for each processor in the re-referenced data chunks.

Figure 16.6 shows the data domain of each processor for the irregular loop in Figure 16.1 before and after executing SW caching phase, with 3 processors and 2 data chunks, where \( x \) is the global array. Before executing the SW caching phase, the Writing area is the one referenced only in the prior loop, and the new Reading area is the one to be referenced only in the later loop. The dark area of \( x \) represents the data referenced in both loops.

In the first Read1 phase, each processor reads data from the file using 2 data chunks. For example, processor P0 reads \( x(2) \sim x(5) \) and distributes \( x(3) \) and \( x(4) \) to P1 and P2 by using a collective I/O method. After finishing the first loop, only data in the Writing Back area is written to the file. The range of the Writing Back area is determined in the Write2 phase. The SW caching phase modifies the schedule information before executing the second loop. Figure 16.6 shows the change on the data domain of each processor after executing the SW caching phase. In the Read2 phase, only data in the New-Reading area is read from the file. Finally, all data referenced in the second loop are written back to the file in the Write1 phase.
16.3.3 Algorithms for the Software Caching Method

We designed and implemented the partial collective I/O operations, read2 and write2, by modifying the basic collective I/O operations. Since the same modifications could be applied for both partial collective I/O operations, we show an algorithm for the partial writing operation, write2. We also show an algorithm for the software caching operation, s/w caching.

To describe the software caching method, let \( C \) be the number of data chunks which will be written in the write2 phase, and \( k \) be the number of incomplete chunks. In other words, some part of a chunk may be re-referenced, but the other part may not be, resulting in the writing of a portion of the chunk, called an incomplete chunk. Let \( l_b1, u_b1, l_b2, \) and \( u_b2 \) be the lower and upper bounds of two irregular loops. Let \( P_i \) be the number of processors which will execute writing or reading operations for the \( i \)-th incomplete chunk. The steps involved in the write2 phase are shown in Figure 16.7.

Let \( RC \) be the number of data chunks in a subsequent loop. Also, let \( SN_{inds}(i,p_d) \) and \( RN_{inds}(i,p_d) \) be the size of the index to be sent and received to/from \( p_i \) in the data chunk \( i \). The steps involved in the software caching phase are shown in Figure 16.8.

16.4 Performance Evaluation

We ran our experiments on a 550 node GBSea with a variety of machine sizes.

1. small
2. medium
3. large
4. extra large

We compared these results against the performance of several existing systems.

Figure 16.6: Data access pattern before and after executing the s/w caching phase.
\textbf{16.4 PERFORMANCE EVALUATION}

We ran our experiments on the Caltech Intel Paragon machine, called \textit{TREX}. \textit{TREX} is a 550 node Paragon XP/S. We performed our experiments using from 32 to 128 compute nodes on 16 and 64 I/O node partitions, each I/O node having 64 MB memory and a 4 GB Seagate disk. The parameters considered for performance results are:

1. number of processors,
2. size of indirection array in the local memory of each processor,
3. data size, and
4. overlap range, which is the data area overlapped between two irregular loops.

We compared the software caching method to the basic collective I/O implementation, called non-caching method, which does not use software caching in the I/O library.

To compare the performance results of both implementations, we classified our evaluation environment by changing the size of the indirection and data arrays, the number of processors, and the degree of data reuse. In each case, we analyzed the performance results of both implementations by separating total execution time into three components: computation time ($t_{\text{comp}}$), communication time ($t_{\text{comm}}$), and I/O time ($t_{\text{i/o}}$). We executed the experiments several times to obtain the average results.
16.4.1 Evaluation Based Upon Phases

We ran both implementations with an indirection array of 512 KB and a data array of 32 MB. The number of processors was 32 and the overlap data area between two loops was 7/8, meaning that 7/8 of the data is shared between the loops. Figure 16.9 shows the execution times for each phase of both implementations. In the s/w caching phase, the computation and communication overheads to reconstruct schedule information are added. The communication time for the write2 phase is larger than for the write1, whereas the I/O time for the write2 phase is smaller than for the write1. The reason is that, in the write2 phase, all processors must communicate with all other processors to distribute their computation results, even though the partial writing operation is only performed by a subset of the processors. We observe that the I/O and communication times for the read2 phase are smaller than for the read1 phase. In the read2 phase, a reduced number of processors perform the partial read operation, and then communicate with others to distribute data. Consequently, even though the computation and communication overheads to reconstruct schedule information are added to the software caching method, we can see that its total execution time is smaller than for the non-caching method.

16.4.2 Evaluation Based Upon the Sizes of Indirection and Data Arrays

In this experiment, we used 32 processors and a 7/8 overlap data area between two loops. We increased the size of the indirection array on each processor’s local memory from 4 KB to 1 MB, resulting in an increment in the data size from 256 KB to 64 MB. Figure 16.10 shows that the total computation time is reduced when the software caching method is applied to the basic collective I/O. The effect of software caching becomes obvious as the size of the indirection array is increased. Fig-
Figures 16.11 and 16.12 show that total communication and I/O times are also reduced when the software caching method is applied. These two figures also show that the benefits of the software caching method in the I/O time are better than in the communication time. As a result, the overall execution time in the software caching method is smaller than in the non-caching method.

16.4.3 Evaluation Based Upon Number of Processors

In this experiment, we executed both implementations with an indirection array of 4 MB and a data array of 64 MB, while changing the number of processors from 32 to 128. Similarly, we fixed the overlap data area between two loops as 7/8. Figure 16.13 represents the execution time divided into the same three components for both implementations. The results show that each component of the total execution time for the software caching method is smaller than for the non-caching method. In particular, when the number of processors is increased from 32 to 128, the computation time is reduced whereas the communication and I/O times are increased. Since the amount of data to be distributed is determined by dividing the global array into the number of processors, the computation time for packing and unpacking data is reduced.

16.4.4 Evaluation Based Upon Overlap Range

In this experiment, we varied the overlap range from 7/8 to 1/8, and then tried to find out the effect of the software caching method. Figure 16.14 shows that the software caching method provides much better performance when data overlap is increased.

When the software caching method is applied to collective I/O, the I/O and computation times are in general less than for the non-caching implementation. There are two reasons for this behavior: the I/O time is reduced in the partial I/O phases when data overlap exists, and the time to compute the data amount to be distributed is reduced. Therefore, the total execution time for the software caching is smaller than for the non-caching implementation when the overlap range is bigger than 2/8 between
two loops. However, when the overlap range is 1/8, the total execution time for the software caching method is greater than the non-caching implementation, because the benefit of the partial I/O operations becomes smaller.

16.5 CONCLUSIONS

In this paper, we presented the design and implementation of the collective I/O with and without the software caching method. The main idea of the software caching method is that the same data may be accessed repeatedly during the execution of subsequent loops in an irregular application. The reused data does not have to be written/read to/from files, thus reducing the amount of I/O data. We also presented the performance results collected on the Caltech Intel Paragon machine. When we change the value of parameters such as the size of indirection (and data) array or the number of processors, the performance results show that, in general, the collective I/O with software caching is better than the collective I/O without software caching.

Acknowledgments

This work was supported in part by Korea Science and Engineering Foundation grants, Sandia National Labs award AV-6193 under the ASCI program, and in part by NSF Young Investigator Award CCR-9357840 and NSF CCR-9509143. This work was developed while all the authors were at Northwestern University.

References


