OPTIMIZING LOCALITY AND PARALLELISM THROUGH PROGRAM REORGANIZATION

DISSERTATION

Presented in Partial Fulfillment of the Requirements for

the Degree Doctor of Philosophy in the

Graduate School of The Ohio State University

By

Sriram Krishnamoorthy, B.E., M.S.

* * * * *

The Ohio State University

2008

Dissertation Committee:

P. Sadayappan, Adviser

Srinivasan Parthasarathy

Atanas Rountev

Approved by

Adviser

Graduate Program in Computer Science and Engineering

ABSTRACT

Development of scalable application codes requires an understanding and exploitation of the locality and parallelism in the computation. This is typically achieved through optimizations by the programmer to match the application characteristics to the architectural features exposed by the parallel programming model. Partitioned address space programming models such as MPI foist a process-centric view of the parallel system, increasing the complexity of parallel programming. Typical global address space models provide a shared memory view that greatly simplifies programming. But the simplified models abstract away the locality information, precluding optimized implementations. In this work, we present techniques to reorganize program execution to optimize locality and parallelism, with little effort from the programmer.

For regular loop-based programs operating on dense multi-dimensional arrays, we propose an automatic parallelization technique that attempts to determine a parallel schedule in which all processes can start execution in parallel. When the concurrent tiled iteration space inhibits such execution, we present techniques to re-enable it. This is an alternative to incurring the pipelined startup overhead in schedules generated by prevalent approaches.

For less structured programs, we propose a programming model that exposes multiple levels abstraction to the programmer. These abstractions enable quick prototyping coupled with incremental optimizations. The data abstraction provides a global view of distributed data organized as blocks. A block is a subset of data stored contiguously in a single process' address space. The computation is specified as a collection of tasks operating on the data blocks, with parallelism and dependence being specified between them. When the blocking of the data does not match the required access pattern in the computation, the data needs to be reblocked to improve spatial locality. We develop efficient data layout transformation mechanisms for blocked multi-dimensional arrays. We also present mechanisms for automatic management of load balance, disk I/O, and inter-process communication on computations expressed as sets of independent tasks on blocked data stored on disk.

To my parents, Sri. S. Krishnamoorthy and Smt. K. Seethalakshmi

ACKNOWLEDGMENTS

This thesis would not have been possible without help from lots of friends and strangers. In addition to being my graduate school stint, these were five years of my life and I would like to thank those that made it memorable. But the list is too long and I will only mention a few.

Sandhya Krishnan and Chi-Chung Lam showed me the methodical approach that motivated me to do my PhD. Sandhya was among the first of a long line of hands that fed me in graduate school. Arjav Chakravarti gave me company on numerous lunches, dinners, and late-nighters. Gerald Sabin introduced me to Slashdot and PhD comics, which probably added a few months to my graduate school. Qingda Lu and Amol Ghoting provided their helpful insights on a lot of things.

There were those attempts at tennis and physical fitness that will keep Rajkiran Panuganti, Muthu Baskaran, Shirish Tatikonda, Uday Bondhugula, and Gaurav Khanna in my memories. Jim Dinan, Joshua Levine, and I made my first snow man. Together with Brian, they have always kept things funny and lively in the lab. My first Thanksgiving experience was with Beth Connell and her family.

My advisor Prof. P Sadayappan provided continuous support, motivation, and enthusiasm throughout my graduate studies. His never-say-die attitude to research has inspired me many a time. I cannot count the number of times I have been told how fortunate I am to have him as my advisor. Srini Parthasaraty and Atanas Rountev provided me timely and useful guidance on research and the publication process. I had insightful discussions with J Ramanujam on loop transformations.

My family made sure I survived through graduate school. I am fortunate to have them as part of my life. My parents inculcated in me a balanced outlook that was extremely helpful through these years. My father's insistence on education and love for books rubbed off on me. He motivated me to pursue graduate school rather than take up a job. My mother constantly reminded me that I need to eat, and is largely the reason I am alive today. I cannot recount all the instances in which my life was made easier by just following my brother's footsteps. He did all the hard work. My sister was the one that kept things light through the years.

If my parents were instrumental in me starting graduate school, Chandrika provided me strong enough encouragement to try to end it. I cannot thank her enough for packing my bags on countless occasions. Despite having known me for the better part of my life, she has agreed to hang around for some more time in the future.

VITA

2002	B.E. Computer Science & Engineering, Anna University, Chennai, TN, India
2006	M.S. Computer Science & Engineering, Ohio State University, Columbus, OH

PUBLICATIONS

X. Gao, S. Krishnamoorthy, S. K. Sahoo, C. Lam, G. Baumgartner, J. Ramanujam, and P. Sadayappan. "Efficient Search-Space Pruning for Integrated Fusion and Tiling Transformations". *Concurrency and Computation: Practice and Experience*, 19(18):2425–2443, December 2007.

S. Krishnamoorthy, G. Baumgartner, C. Lam, J. Nieplocha, and P. Sadayappan. "Layout Transformation Support for the Disk Resident Arrays Framework". *Journal of Supercomputing*, 36(2):153–170, May 2006.

A. Auer, G. Baumgartner, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Krishnamoorthy, S. Krishnan, C. Lam, M. Nooijen, R. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. "Automatic Code Generation for Many-Body Electronic Structure Methods: The Tensor Contraction Engine". *Molecular Physics*, 104(2):211–228, January 2006.

S. Krishnan, S. Krishnamoorthy, G. Baumgartner, C. Lam, J. Ramanujam, P. Sadayappan, and V. Choppella. "Efficient Synthesis of Out-of-Core Algorithms Using a Nonlinear Optimization Solver". *Journal of Parallel and Distributed Computing*, 66(5):659–673, May 2006.

G. Baumgartner, A. Auer, D.E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R.J. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C. Lam, Q. Lu, M. Nooijen, R.M. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. ”Synthesis of High-Performance Parallel Programs for a Class of Ab Initio Quantum Chemistry Models”. *Proceedings of the IEEE*, 93(2):276–292, February 2005.

S. Krishnamoorthy, G. Baumgartner, D. Cociorva, C. Lam, and P. Sadayappan. “Efficient Parallel Out-of-core Matrix Transposition”. *International Journal of High Performance Computing and Networking*, 2(2/3/4):110–119 2004.

M. Baskaran, Uday Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. “Automatic Data Movement and Computation Mapping for Multi-level Parallel Architectures with Explicitly Managed Memories”. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP’08)*, February 2008.

S. Krishnamoorthy, J. P. Canovas, V. Tipparaju, J. Nieplocha, and P. Sadayappan. “Non-Collective Parallel I/O for Global Address Space Programming Models”. In *Proceedings of Cluster 2007*, September 2007.

S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. “Effective Automatic Parallelization of Stencil Computations”. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’07)*, June 2007.

S. Krishnamoorthy, U. Catalyurek, J. Nieplocha, and P. Sadayappan. “Hypergraph Partitioning for Automatic Memory Hierarchy Management”. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. November 2006 (SC 2006)*, November 2006.

M. Blocksome, C. Archer, T. Inglett, P. McCarthy, M. Mundy, J. Ratterman, A. Sidelnik, B. Smith, G. Almasi, J. Castanos, D. Lieber, J. Moreira, S. Krishnamoorthy, and V. Tipparaju. “Design and Implementation of a One-Sided Communication Interface for the IBM eServer Blue Gene Supercomputer”. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. November 2006 (SC 2006)*, November 2006.

N. Vydyanathan, S. Krishnamoorthy, G. Sabin, U. Catalyurek, T. Kurc, P. Sadayappan, and J. Saltz. “Locality Conscious Processor Allocation and Scheduling for Mixed-Parallel Applications”. In *Proceedings of the IEEE International Conference on Cluster Computing*, September 2006.

Q. Lu, S. Krishnamoorthy, and P. Sadayappan. “Combining Analytical and Empirical Approaches in Tuning Matrix Transposition”. In *Proceedings of the 15th International Conference on Parallel Architectures and Compiler Techniques. (PACT 2006)*, September 2006.

N. Vydyanathan, S. Krishnamoorthy, G. Sabin, U. Catalyurek, T. Kurc, P. Sadayappan, and J. Saltz. “An Integrated Approach for Processor Allocation and Scheduling of Mixed-Parallel Applications”. In *Proceedings of the 35th International Conference on Parallel Processing (ICPP 2006)*, August 2006.

A. Hartono, Q. Lu, X. Gao, S. Krishnamoorthy, M. Nooijen, G. Baumgartner, V. Choppella, D. E. Bernholdt, R. M. Pitzer, J. Ramanujam, A. Rountev, and P. Sadayappan. “Identifying Cost-Effective Common Subexpressions to Reduce Operation Count in Tensor Contraction Evaluations”. In *Proceedings of the 6th International Conference on Computational Science (ICCS 2006)*, May 2006.

G. Khanna, N. Vydyanathan, U. Catalyurek, T. Kurc, S. Krishnamoorthy, P. Sadayappan, and J. Saltz. “Task Scheduling and File Replication for Data-Intensive Jobs with Batch-Shared I/O” In *Proceedings of the 15th IEEE International Symposium on High Performance Distributed Computing (HPDC 2006)*, June 2006.

S. Krishnamoorthy, J. Nieplocha, and P. Sadayappan. “Data and Computation Abstractions for Dynamic and Irregular Computations”. In *Proceedings of the 12th Annual International Conference on High Performance Computing (HiPC 2005)*, December 2005.

S. K. Sahoo, S. Krishnamoorthy, R. Panuganti, and P. Sadayappan. “Integrated Loop Optimizations for Data Locality Enhancement of Tensor Contraction Expressions” In *Proceedings of Supercomputing (SC 2005)*, November 2005.

S. K. Sahoo, R. Panuganti, S. Krishnamoorthy, and P. Sadayappan. “Cache Miss Characterization and Data Locality Optimization for Imperfectly Nested Loops on Shared Memory Multiprocessors”. In *Proceedings of the 19th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2005)*, April 2005.

S. Krishnamoorthy, G. Baumgartner, C. Lam, J. Nieplocha, and P. Sadayappan. “Efficient Layout Transformation Support for Disk-based Multidimensional Arrays”. In *Proceedings of the 11th Annual International Conference on High Performance Computing (HiPC 2004)*, December 2004.

S. Krishnan, S. Krishnamoorthy, G. Baumgartner, C. Lam, J. Ramanujam, P. Sadayappan, and V. Choppella. “Efficient Synthesis of Out-of-Core Algorithms Using a Nonlinear

Optimization Solver”. In *Proceedings of the 18th International Parallel & Distributed Processing Symposium (IPDPS 2004)*, April 2004.

S. Krishnan, S. Krishnamoorthy, G. Baumgartner, D. Cociorva, C. Lam, P. Sadayappan, J. Ramanujam, David E. Bernholdt, and V. Choppella. “Data Locality Optimization for Synthesis of Efficient Out-of-Core Algorithms”. In *Proceedings of the 10th Annual International Conference on High Performance Computing (HiPC 2003)*, December 2003.

S. Krishnamoorthy, G. Baumgartner, D. Cociorva, C. Lam, and P. Sadayappan. “Efficient Parallel Out-of-core Matrix Transposition”. In *Proceedings of IEEE International Conference on Cluster Computing (CLUSTER 2003)*, December 2003.

FIELDS OF STUDY

Major Field: Computer Science and Engineering

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

CHAPTER 1

INTRODUCTION

There have been dramatic strides in hardware performance of modern high-end systems. These improvements have been accompanied by a corresponding increase in the complexity of these systems. Modern parallel computers have increasingly large number of processors, deeper memory hierarchies, and higher processor clock speeds. This has resulted in an increasingly important role of exploitation of parallelism in the computation to utilize the available processors, and data locality to maximize the time spent in useful computation by processors.

The dramatic strides in hardware performance of modern high-end systems over the past decades have not been matched by a corresponding improvement in the ease of programming them. The increasingly complex hardware and communication architectures, while enabling high performance, have resulted in an increasing amount of detail being handled by the programmer to achieve that performance.

From a programmer's viewpoint, the complexity of the code required to implement a given algorithm or simulation is a function of the level of detail the programming model exposes to the programmer, the number of decisions and choices to be made, together with the level of detail required to manage performance-related aspects of the underlying hardware.

Developing scalable application codes on such systems requires an understanding and exploitation of the parallelism and locality in the computation. The computation needs to be parallelized such that good load balance is achieved. The distribution of the data amongst the processors needs to take into account the cost of data movement between the processors. In addition, movement of data between the different levels of the memory hierarchy needs to minimized to reduce the overall data movement cost. This is usually achieved by distinguishing the different levels of the memory hierarchy that incur different data movement costs. The programmer optimizes the execution of an application by matching the locality and parallelism identified in the application to the architectural abstraction exposed by the parallel programming model.

Parallel programming models provide a combination of data and control abstractions. The data abstraction divides the addressable memory in the system into units of locality. All elements in the same unit of locality are assumed to incur the same data movement cost. A simplified data abstraction that improves productivity ideally presents a uniform view of the memory in the system. On the other hand, achieving good scalability requires a data abstraction that can be efficiently mapped to the non-uniform deep memory hierarchy in modern parallel systems. This is typically achieved by creating one unit of locality for each processor memory, and distinguishing between access to the data in local memory and to that in non-local memory.

The control abstraction provides mechanisms to identify and express the parallelism in the computation. It also optionally provides mechanisms for automatic load balancing. The abstraction is either computation-centric, with parallelism specified in terms of data or functions used in the computation, or architecture-centric. In an architecture-centric control abstraction a fixed number of control flow units are defined, typically one per processor, and these are allowed to execute in parallel with user managing the synchronization between them. While an architecture-centric control abstraction can be easily mapped to the processors in a parallel system, computation-centric abstractions facilitate ease of programming. In addition, architecture-centric specification of parallelism encourages the programmer to partition the computation with implicit dependences between the tasks in a part. The absence of this information to the runtime makes automatic support for load-balancing a challenging task.

When an application's data does not fit into the collective physical memories of a parallel system, the data is stored on disk. Such data is referred to as *out-of-core* data. Virtual memory has been shown to inefficient in handling out-of-core scientific applications due to a lack of insights into the data access characteristics in the application. Explicit disk I/O mechanisms are used to move the data between the secondary storage and main memory. In programming out-of-core applications, a programmer has to contend with the orchestration of the movement of data between disk and memory, ensuring that the memory utilization does not exceed the size of the available physical memory, and scheduling the computation to operate only on the data in the physical memory. In computations based on the traditional collective disk I/O model, all processes collective move the data between the disks and the distributed memory, with each process subsequently performing communication to move the data into the local memory for processing.

We attempt to improve the performance of an application with limited impact on productivity. For regular loop-based programs operating on dense multi-dimensional arrays, we present an automatic parallelization technique that enables the concurrent start of execution by all processors. This avoids the pipelined startup overhead incurred by schedules generated by prevalent approaches. It also exposes the trade-off between communication volume, the number of communication start-ups, or computation cost and the parallelism in the application. Unlike existing loop transformation frameworks, this approach enables the user to choose between the various costs depending on the application and the balance of computation and communication costs in the target system. Chapter 2 presents the technique and evaluates the approach on stencil codes.

For less structured programs, we propose to reconcile the seemingly conflicting requirements of performance and productivity by presenting the user with multiple interoperable control and data abstractions, each at a different level of detail. This enables a programmer to realize an implementation of an algorithm using a high level abstraction, and incrementally optimize it improve its efficiency and scalability.

A key aspect of our approach to handle parallelism and data locality, including out-of-core data, is the organization of the data into blocks that are globally addressable. A block is a subset of data that is stored in a single processor's memory or disk. A block is defined to be the basic unit of locality. The computation is defined in terms of sets of tasks operating on the blocks of data and dependences between them.

When the layout of the blocks does not match the access pattern in the computation, a performance penalty is incurred due to the lack of spatial locality. This is particularly severe in the context of out-of-core data. We present efficient data layout transformation algorithms to transform the blocking of multi-dimensional arrays, when the blocking required for different phases are very different. We first develop an out-of-core matrix transposition algorithm that takes into account the I/O characteristics of the target system. We then present a novel algorithm to solve the out-of-core matrix reblocking problem for multi-dimensional matrices of arbitrary sizes. This is topic of discussion in Chapter 3.

The programmer specifies a task as a set of data blocks and a function that contains an efficient sequential implementation of an algorithm to process those blocks. The blocking of data ensures efficient data movement, while the tasks provide efficient sequential executions to process interacting blocks. Blocking of data thus enables us to leverage existing work on optimizing sequential computations.

This abstraction provides a computation-centric view of both locality and parallelism, thus presenting a simple abstraction to the user. On the other hand, the explicit specification of the locality and parallelism enables runtime mechanisms that map the data and tasks to the processors in a parallelism to automatically manage locality, dependences, and load balance in the execution of the program. We believe these features can enable the presentation of a high-level programming abstraction without compromising scalability and sequential efficiency.

We demonstrate the benefits of the approach in the context of tensor contractions arising from the quantum chemistry domain in Chapter 4. We present mechanisms for automatic management of load balance, disk I/O, and inter-process communication on the quantum chemistry computations expressed as sets of independent tasks on blocked data stored on disk. The objective is to minimize the volume of disk I/O while balancing the computation amongst the processors.

CHAPTER 2

EFFECTIVE AUTOMATIC PARALLELIZATION OF STENCIL COMPUTATIONS

2.1 Introduction

Stencil computations represent a practically important class of computations that arise in many scientific/engineering codes. Computational domains that involve stencils include those that use explicit time-integration methods for numerical solution of partial differential equations (e.g., climate/weather/ocean modeling [100], computational electromagnetics codes using the Finite Difference Time Domain method [108]), and multimedia/image-processing applications that perform smoothing and other neighbor pixel based computations [45]. There has been some prior work from the computer science community that has addressed performance optimization of stencil computations (e.g., [99, 54, 53, 39]). Since stencil computations are characterized by a regular computational structure, they are amenable to automatic compile-time analysis and transformation for exploitation of parallelism and data locality optimization. However, as elaborated later through an example, existing compiler frameworks have limitations in generating efficient code optimized for parallelism and data locality.

Loop tiling is the key transformation to enable parallelization and data-locality optimization of stencil codes. Much research has been published on tiling of iteration spaces [52, 119, 118, 106, 29, 34, 92, 95, 49, 14, 50, 33, 51, 3]. With few exceptions (e.g., work of Griebl [42, 43]), research on performance optimization with tiling has generally focused on one or the other of the two complementary aspects: (a) data locality optimization [4, 3, 118, 106, 29]; or (b) tile size/shape optimization for parallel execution [34, 92, 9, 49, 14, 50, 33, 51]. Tiling for data locality optimization involves maximization of data reuse, i.e., tiling along directions of the data dependence vectors. But such tiling may result in inter-tile dependences that inhibit concurrent execution of tiles on different processors. To the best of our knowledge, no prior work has addressed in an integrated fashion, the issues of tiling for data locality optimization and load balancing for parallel execution. We first use the simple example of a one-dimensional Jacobi code to illustrate the problem and introduce two approaches we propose to avoid the problem: overlapped tiles and split tiles. As an example of a stencil computation, let us consider the one-dimensional Jacobi code shown in Figure 2.1. Optimizing this stencil computation for reduction of cache misses requires loop fusion and tiling; in order to fuse the two inner loops, loop skewing is needed. Frameworks have been previously proposed for data locality optimization of imperfectly nested loops. Using an approach proposed by Ahmed et. al. [3, 4] the loop nest can be transformed into the one shown in Figure 2.2 by first embedding the iterations in the imperfectly-nested loops into a perfectly-nested iteration space. Loop transformations and tiling can then be applied in the transformed perfectly-nested iteration space. The transformed iteration space can be subsequently translated into efficient code by reducing/eliminating the control overhead [60]. In this chapter, we focus on

```
   for t = 0 to T-1
     for i = 1 to N-1
S1:    B[i] = (A[i-1]+A[i]+A[i+1])/3
     for i = 1 to N-1
S2:    A[i] = B[i]
```

Figure 2.1: Imperfectly-nested one-dimensional Jacobi

load-balanced parallel execution of tiled iteration spaces that have already been embedded into a perfectly-nested iteration space using a technique such as that developed in [4].

Figure 2.3 shows a single-statement form of the one-dimensional Jacobi code obtained by adding an additional dimension to array A. The flow dependences in this code are the same as that of the previously shown version, but there are no anti-dependences. Hence a single statement is sufficient in the loop body instead of a sequence of two statements, for update and copy, respectively, as seen in Figures 2.1 and 2.2. Although such a memory-inefficient code would not be used in practice, it is more convenient to use a single-statement iteration space in explaining the main ideas in this chapter. However, the developed approach is not restricted to such single-statement loops, but is applicable to general multi-statement stencil codes such as the one in Figure 2.1. The experimental results presented later also use the memory-efficient multi-statement versions.

The perfect loop nest of Figure 2.3 has constant dependences $(1,0)$, $(1,1)$, and $(1,-1)$. Tiling for data reuse optimization (e.g. using the approach presented in [2]) results in tiles of shape as shown in Figure 2.4. The horizontal axis corresponds to the spatial dimension, with time along the vertical dimension. Using a sufficiently large tile size along the time dimension facilitates significant data reuse within caches/registers. However, there are inter-tile dependences in the horizontal direction, inhibiting concurrent execution of tiles

```
   for t = 0 to T-1
     for i = 1 to N
       if(i>=1 and i<=N-1)
S1:      B[i] = (A[i-1]+A[i]+A[i+1])/3
       if(i>=2 and i<=N)
S2:      A[i-1] = B[i-1]
```

Figure 2.2: Fused one-dimensional Jacobi

by different processors. However, if the vertical tile size is reduced to one (i.e., tiling is eliminated along the time dimension), all tiles along the spatial dimension (adjoining the x-axis) can be executed concurrently. Thus there is a trade-off between achieving good data reuse and load balancing of parallel execution.

Instead of the *standard* tiling described above, consider the tiling shown in Figure 2.5. Starting with the tiles formed by the same hyperplanes, an additional triangular region is added to the left of the tile, overlapping with the points at the right end of the neighboring tile. With this tiling, the iteration points processed by the tiles are no longer disjoint. Some of the iterations are executed redundantly by two neighboring tiles. This results in an increase in the computation cost. But doing so eliminates the dependence between tiles along the horizontal direction. All processors can start executing in parallel, eliminating the initial processor idling that results with the pipelined parallel execution of tiles in Figure 2.4.

While standard tiling can enhance data locality in this context, overlapped tiling can both improve data locality and eliminate the overhead of pipelined parallelism, at the cost

```
for t = 0 to T-1
  for i = 1 to N-1
    A[t,i] = (A[t-1,i-1] + A[t-1,i] + A[t-1,i+1])/3
```

Figure 2.3: Single-statement form of one-dimensional Jacobi

of slightly increased computation time. However, the increased computational cost is independent of tile size. Therefore the fractional computation overhead is inversely proportional to the tile size in the direction of overlapped tiling, and can be made insignificant if a sufficiently large tile size is chosen along the time dimension.

An alternate approach, shown in Figure 2.6, splits the interior of each tile into two sub-tiles, where the points in only one of the two sub-tiles (shaded) are dependent on points in the neighbor tile, while the points in the other sub-tile are not dependent on any neighboring tile's points, and therefore executable concurrently. With this approach, each standard tile is split into two sub-tiles, and load-balanced concurrent execution is possible as a sequence of two steps: first all non-dependent sub-tiles are concurrently executed and communicate with the neighbor tiles, and then the dependent sub-tiles are all concurrently executed.

The chapter is organized as follows. Section 2 defines the problem addressed in this chapter. In Section 3, we characterize the conditions under which tiled iteration spaces can benefit from overlapped/split tiling. In Section 4, we show how to transform a given tiled iteration space in order for overlapped/split tiling to be applicable. Section 5 discusses code generation and Section 6 analyzes the cost benefits of overlapped tiling. Section 7 provides experimental results that demonstrate the benefits of overlapped/split tiling. In Section 8, we discuss related work and conclude in Section 9 with a summary.

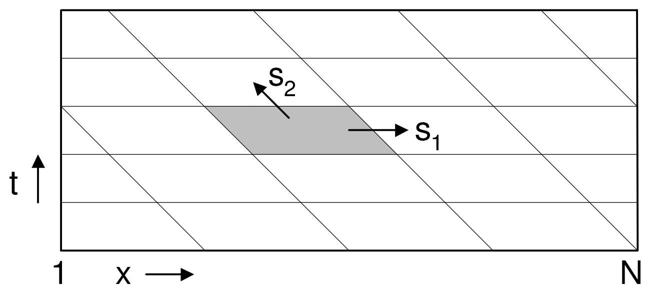


Figure 2.4: Standard tiling for one-dimensional Jacobi. s_1 and s_2 denote the inter-tile dependences.

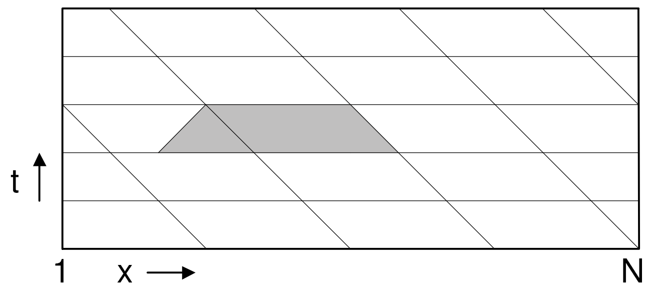


Figure 2.5: Overlapped tiling for one-dimensional Jacobi

2.2 Background and Problem Statement

This section introduces some standard background on the polyhedral model of computation, and defines the problem addressed. Consider a perfectly-nested loop nest with n levels of nesting. The *iteration space polyhedron* defines an n-dimensional set of points, characterized by a set of bounding hyperplanes and modeled as $B.I \geq b$ where I is the

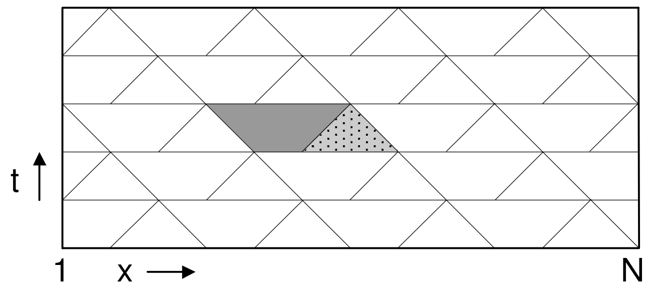


Figure 2.6: Split tiling for one-dimensional Jacobi

iteration vector. The rows b_i of B define the normals to the corresponding bounding hyperplanes. For example, the iteration space for the 1-D Jacobi example is

$$\begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} t \\ i \end{pmatrix} \geq \begin{pmatrix} 0 \\ -T+1 \\ 1 \\ -N+1 \end{pmatrix}$$

The dependences in the computation can be represented by a matrix D where each column defines a dependence vector. The dependences in the 1-D Jacobi example are

$$D = \begin{pmatrix} d_1 & d_2 & d_3 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ -1 & 0 & 1 \end{pmatrix}$$

Assume that we are given a set of *tiling hyperplanes* that tile the iteration space. These hyperplanes are represented by a matrix H, where each row represents the normal vector of a tiling hyperplane. For example, the tiling hyperplanes corresponding to Figure 2.4 are represented as

$$H = \begin{pmatrix} h_1 \\ h_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

A tiling defined by a set of tiling hyperplanes is *legal* if each tile can be executed atomically and there exists a valid total ordering of the tiles. Intuitively, a tiling is legal if no two tiles mutually depend on each other. It can be shown [52] that this validity condition is given by

$$H.D \geq 0$$

A schedule has a *concurrent start* property if all processors can start execution in parallel, without any pipeline start-up delay. Such a schedule is referred to as a concurrent-start schedule.

Problem Statement. Consider a given (non-tiled) iteration space in which a concurrent-start schedule is possible. However, for a given tiling of this space defined by a set of tiling hyperplanes, it is possible that the tile dependencies in the corresponding tiled iteration space inhibit concurrent start. We consider the following question: How can concurrent start be achieved in the tiled iteration space? Our first goal is to characterize analytically the situations in which tiling inhibits concurrent start. Next, we develop two approaches, *overlapped* tiling and *split* tiling, that enable concurrent start in the tiled space and recover the load-balancing properties lost due to tiling.

2.3 Inhibition of Concurrent Start

If the original non-tiled iteration space does not have a concurrent start schedule, tiling cannot enable such a schedule. However, if concurrent start is possible in the absence of tiling, the introduction of tiling can potentially inhibit this concurrent start. This section characterizes the conditions under which a non-tiled space supports a concurrent start schedule, and then derives a concurrent start inhibition condition for the tiled space. For simplicity of presentation, the discussion assumes an iteration space with a single statement, but we have defined a general version of the technique for multi-statement iteration spaces.

2.3.1 Concurrent Start in the Non-Tiled Space

First, we describe the condition for the existence of concurrent start in the original non-tiled iteration space. Consider, for example, dependence vectors $(1, 0)$ and $(0, 1)$. Two

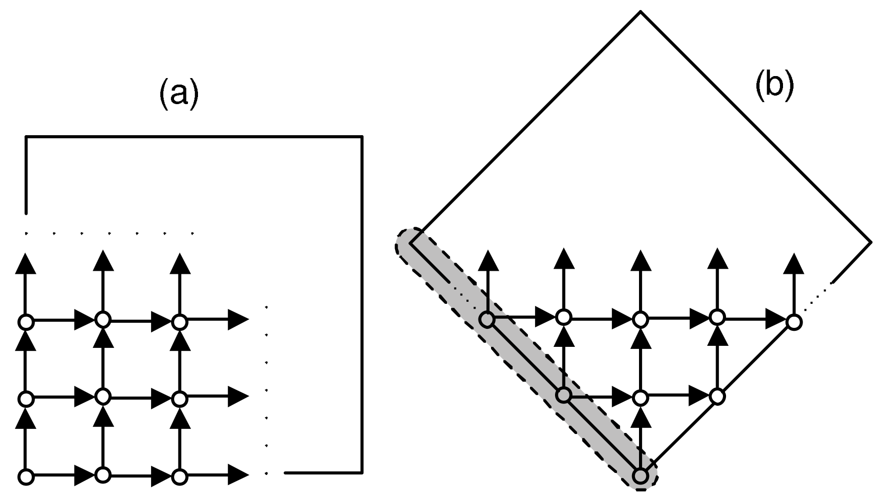


Figure 2.7: Illustration of concurrent-start. Iteration spaces with $(1,0)$ and $(0,1)$ dependencies: (a) concurrent start is not possible (b) concurrent start is possible from the gray boundary.

iteration spaces with these dependences are shown in Figure 2.7. In Figure 2.7(a), the parallel computation has to begin from the origin $(0,0)$ and suffers from pipeline start-up overhead. On the other hand, the iteration space in Figure 2.7(b) can be traversed by all processors in parallel starting from the boundary shown in gray.

In general, the presence of concurrent start in an iteration space depends on the boundaries that define the iteration space polyhedron. An iteration space supports concurrent start if there exists a bounding hyperplane that does not contain a dependence, i.e. carries all dependences. A hyperplane contains a dependence if both the source and destination iteration points of the dependence are contained in the hyperplane. Since the rows b_i of B define the normal vectors of the bounding hyperplanes, this property is represented by the condition

$$\exists b_i \in B \;:\; \forall d_j \in D \;:\; b_i.d_j > 0$$

Note that this condition is independent of the tiling hyperplanes. We will refer to this property as the *point-wise concurrent start condition*. When this condition does not hold, no tiled iteration space can have concurrent start. For the 1-D Jacobi example, the condition holds because the normal vector $b_1 = (1\ \ 0)$ for one of the bounding hyperplanes satisfies $b_1.d_j > 0$ for all dependence vectors d_j.

2.3.2 Inhibition of Concurrent Start in the Tiled Space

Next, we consider the condition for the inhibition of the concurrent start condition in the tiled iteration space. Given the tiling hyperplanes and their normal vectors $h_i \in H$, we define the *shift vector* s_i for the hyperplane with h_i as normal to be a vector connecting two instances of the same hyperplane, while traveling parallel to all other hyperplanes. Clearly, the following holds for the set S of shift vectors:

$$\forall s_i \in S\ :\ \forall j \neq i\ :\ h_j.s_i = 0$$

For the 1-D Jacobi example, we will use shift vectors

$$S = \left(\begin{array}{cc} s_1 & s_2 \end{array}\right) = \left(\begin{array}{cc} 0 & 1 \\ 1 & -1 \end{array}\right)$$

as illustrated in Figure 2.4.

The execution of two adjacent tiles should be ordered if there is a dependence vector d_j such that for some iteration points i_1 and i_2 related by d_j, point i_1 is in one of the tiles and point i_2 is in the other one. Note that this is possible only if there is a dependence that passes through the hyperplane that separates the two tiles — in other words, if the following condition holds

$$\exists d_k \in D\ :\ h_i.d_k \neq 0$$

When this condition is satisfied for a given hyperplane with $h_i \in H$, the shift direction s_i along that dimension *carries the inter-tile dependence*. For the 1-D Jacobi example, both s_1 and s_2 carry inter-tile dependencies (for example, $h_1.d_1 > 0$ and $h_2.d_1 > 0$).

The inter-tile dependences can introduce dependence directions that do not exist in the original iteration space. The concurrent start condition is *inhibited in the tiled iteration space*, if for some boundary b_i, the concurrent start condition is satisfied by the dependences in the original iteration space, but not by the inter-tile dependences in the tiled iteration space. A tiling inhibits concurrent start if

$$\exists b_i \in B, h_j \in H, d_k \in D : b_i.D > 0 \wedge b_i.s_j = 0 \wedge h_j.d_k \neq 0$$

When the above condition is true, there exists an inter-tile dependence within a hyperplane parallel to the boundary b_i, precluding concurrent execution of all the tiles in the boundary. Thus, concurrent start is inhibited even though the original iteration space supports it. This situation occurs for the 1-D Jacobi example due to bounding plane normal $b_1 = (1 \;\; 0)$, tiling hyperplane normal $h_1 = (1 \;\; 0)$, and any dependence d_k for $k = 1 \ldots 3$.

2.4 Overlapped Tiling

The basic idea behind overlapped tiling is to eliminate certain inter-tile dependencies by “duplicating” points in the original iteration space. As a result, the same iteration point can be a member of two neighboring tiles (i.e., the tiles can overlap). This section outlines a constructive procedure to determine overlapping tiles that eliminate the inter-tile dependences, which removes the inhibition on concurrent start. The key step is the construction of a *companion hyperplane* that eliminates the dependence along a desired direction. The new tile will not have any incoming dependence along the direction in which the dependence was eliminated.

In standard tiling, a hyperplane with a normal vector h_i defines two faces of the tile. We will denote these faces as $h_i(l)$ (the back face) and $h_i(l+1)$ (the front face). The front face is shared with the subsequent tile along the shift direction defined by shift vector s_i. The back face $h_i(l)$ has no incoming dependences if $h_i.D \geq 0$. On the other hand, the front face $h_i(l+1)$, by the tiling validity condition, does not have any incoming dependences. All dependences between the hyperplanes can be eliminated if the back face of the tile is replaced by an overlapped hyperplane with a normal vector h_i' such that

$$\forall d_j \in D : h_i'.d_j \leq 0$$

Note that the hyperplanes span the iteration space and any vector in the iteration space; hence, the companion hyperplane can be defined as a linear combination of the existing hyperplanes. Scaling a given hyperplane vector h_i does not eliminate any additional dependences. In addition, we are interested in the companion hyperplane that forms the back face of the tile. Thus, it is constructed by going "backwards" on the other hyperplanes, represented by a negative linear combination of the hyperplanes, and is given by:

$$h_i.D \geq 0 \Rightarrow h_i' = h_i - \sum_{j \neq i} k_j.h_j \wedge h_i'.D \leq 0 \wedge k_j > 0$$

Such a companion hyperplane eliminates dependences along a shift vector. This procedure is repeated for every hyperplane/shift vector that inhibits concurrent start.

Consider n-dimensional Jacobi iteration with an $n+1$ dimensional iteration space, and an n dimensional data space, with a range of N along each dimension. Let B be the space tile size along each of the n space dimensions. Let p be the number of processors organized in an n-dimensional grid. $B = N/\sqrt[n]{p}$. Let t be the time tile size.

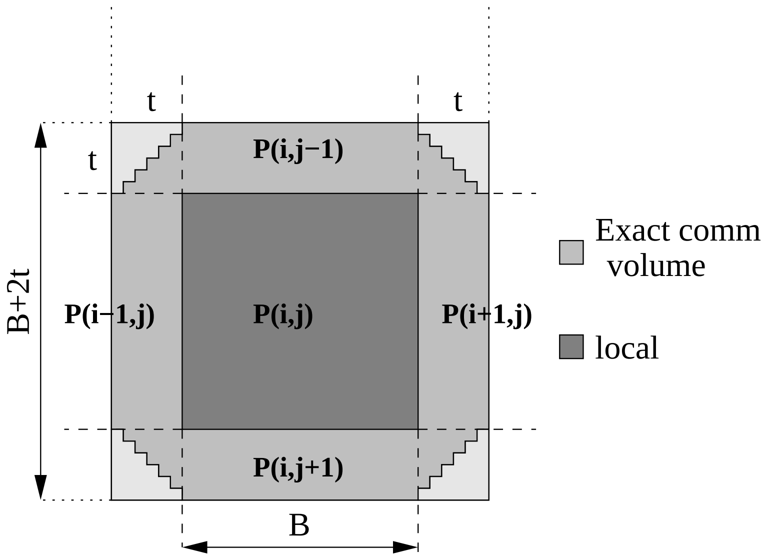


Figure 2.8: Overlapped tiling for two-dimensional Jacobi: top view

The schedule for overlapped tiling requires the processors to cycle to maintain load balance. We illustrate the determination of communication frequency using a simpler variation. Starting from orthogonal tiling, both planes can be swiveled partially to form trapezoid-like tiles for 1-D Jacobi, and a square pyramid for two-dimensional Jacobi, top view for which is shown in Figure 2.8. This overlapped tiling scheme has the same communication volume as the original one, but double the number of startups. However, code generation is simpler for this case due to the absence of the need to cycle. The number of startup's do not matter when the communication volume is higher; this is particularly true for higher dimensional Jacobi (greater than 1) for which the space tile size comes into the volume.

Consider the overlapped tiling scheme that is obtained from orthogonal tiling. The point-wise difference between the coordinates of a given processor and any of its neighbors in the processor space is an n-vector, and each of its n components being 1, 0, or -1.

Discounting the all zeros case, we have $3^n - 1$ neighbors. Hence, the number of communication startups per tile (without forwarding) is given by:

$$S_1 = 3^n - 1 \tag{2.1}$$

For example, for three-dimensional Jacobi, we have 8 corners, 12 edges, and 6 faces, i.e., a total of 26 ($= 3^3$-1) neighbors to send and receive data to/from to compute the overlapped tile.

With communication forwarding, the number of communication startups per tile can be reduced to $2n$ (one for each of the faces).

$$S_1' = 2n \tag{2.2}$$

Similarly the number of startups for the original schedule without and with forwarding are:

$$S_2 = 2^n - 1 \tag{2.3}$$

$$S_2' = n \tag{2.4}$$

The exact communication volume assuming orthogonal tiling is given by:

$$\begin{aligned} V &= \sum_{i=1}^{n} \binom{n}{n-i} 2^i B^{(n-i)} f(i,t) \\ &\approx 2ntB^{n-1} \text{ when } t \ll B \end{aligned} \tag{2.5}$$

where

$$f(k,t) = \sum_{i_{n-k+1}=1}^{t-1} \cdots \sum_{i_n=1}^{i_{(n-1)}} i_n \tag{2.6}$$

The communication volume for the original schedule reduces to:

$$\begin{aligned} V &= \sum_{i=1}^{n} \binom{n}{n-i} B^{(n-i)} f(i,2t) \qquad (2.7) \\ &\approx 2ntB^{n-1} \text{ when } t \ll B \end{aligned}$$

The communication schedule and the data being communication can be quite complex for higher dimensions. Adding a small number of points to the communication volume greatly simplifies code generation. In Figure 2.8, the points in each of the four corners are those that can be added. The total communication volume then becomes:

$$\begin{aligned} V' &= (B+2t)^n - B^n \\ &= {}^nC_1 B^{n-1}(2t) + {}^nC_2 B^{n-2}(2t)^2 \\ &\quad + \cdots + (2t)^n \qquad (2.8) \\ &\approx 2ntB^{n-1} \text{ if } t \ll B \\ &= \Theta(tB^{n-1}) \qquad (2.9) \end{aligned}$$

For $n = 2$:

$$V' = 4tB + 4t^2$$

2.5 Split Tiling

Overlapped tiling eliminates inter-tile dependences by redundantly computing portions of a tile. While eliminating dependences, this approach increases the overall amount of computation. In this section we leverage the idea of dependence inhibition to develop an alternative approach, referred to as *split tiling*, in order to enable concurrent start without the computation overhead. In split tiling, rather than redundantly computing a portion of the predecessor tile along a dimension, the processor executing the predecessor tile first computes that portion and sends the results to its successor along that dimension.

We show that for stencil computations, a tile sub-region can be identified such that this sub-region can be executed in parallel in all tiles. This enables concurrent start. We outline an algorithm that divides a tile into sub-regions and schedules the computation and communication to achieve concurrent start and load-balanced execution in which all processors execute the same amount of work in all the steps in the schedule.

A tile in a stencil computation is bounded by the hyperplane instances:

$$\forall I, B.I \geq b, h_j \in H : h_j.I \geq lo_j, h_j.I \leq hi_j$$

where two parallel instances of each hyperplane are defined, one bounding the tile below along that dimension and another bounding the tile from above.

Along a dimension j, dependence inhibition identifies a partner hyperplane such that the region enclosed by the partner hyperplane (h'_j) in the positive direction ($h'_j.I \geq lo'_i$ can be computed independently of the rest of the tile. This region was redundantly computed in the overlapped tiling approach.

Definition 1. *The independent region along a dimension j is denoted by $\neg j$. The rest of the tile along that region will be denoted by j.*

In the subsequent discussion, it should be clear from the context whether j refers to the dimension or to the complement of the independent region along that dimension.

The region $\neg j$ is defined by making the partner hyperplane to be bounded from below along that dimension:

$$\forall I, B.I \geq b, h_k \in H, k \neq j : h_k.I \geq lo_k, h_k.I \leq hi_k$$

$$\forall I, B.I \geq b : h'_j.I \geq lo'_k, h_j.I \leq hi_j$$

Note that the hyperplanes along all the other dimensions remain unchanged.

A tile can be divided into these two regions along each of the dimensions. The various intersections of these regions divides the tile into 2^k tile components for k such dimensions. We only consider dimensions along which there is potential for dependence inhibition, which would eliminate the time dimension. For example, a tile in the 2-D Jacobi code with x and y as the dimensions can be divided into the components $\neg x \cap \neg y$, $\neg x \cap y$, $x \cap \neg y$, and $x \cap y$.

From the definition of independent region, a tile component $\neg i \cap \ldots$ is not dependent on its predecessor along dimension i. Thus, the tile component that is the intersection of the independent tile region along all the processors can be computed in parallel, without any communication — that is, all processors can start executing this in parallel, resulting in concurrent start.

Consider the tile component $i \cap \ldots$, where all other tile regions are independent. This tile component does not carry any dependence along any dimension other than i. The region in the predecessor tile that it depends on is derived as the tile-component with the same hyperplanes along all other dimensions as the tile component, with the hyperplanes along dimension i replaced by the lower-bounding hyperplane for this tile becoming the upper-bounding hyperplane, and the partner hyperplane for dependent inhibition becoming the lower-bounding hyperplane. This is the tile component $\neg i \cap \ldots$. Thus, the tile component $i \cap \ldots$ can be computed once the boundary along i computed by $\neg i \cap \ldots$ in the predecessor tile.

In general, for each dimension i along which a tile component is dependent, the inter-tile boundary is computed by the tile component in the predecessor tile obtained by replacing i by $\neg i$ For example, the tile component $x \cap y$ in the 2-D Jacobi code can be computed after

```
1: If ($n$==1), say a dimension $x$. Compute $\neg x$, send and receive the result along the $x$ dimension,
       compute $x$ and return.
2: Execute algorithm for (n-1)-dimensional stencil computation for all dimensions except one, say
       z. Thus all values computed will be for those independent along $z$ (all tile sections have $\neg z$
       as the z dimension component).
3: Send all computed values along the $z$ dimension.
4: Execute algorithm for n-dimensional stencil computation for all dimensions except z. But this
       time, all values computed will be dependent for dependent regions along $z$.
```

Algorithm 2.1: Computation/communication scheduling algorithm for split-tiling

the shared boundary with $\neg x \cap y$ is received from the predecessor along x, and the one with $x \cap \neg y$ is received from the predecessor along y.

Algorithm 2.1 presents a scheduling algorithm with $2^n - 1$ communication steps for an n-dimensional stencil computation. In this recursive formulation, the number of communication steps is given by :

$$L(n) = 2 * L(n-1) + 1$$

with L(1)=1; that is, $L(n) = 2^n - 1$. Note that this approach does not incur any addition computation cost. In addition, only inter-tile boundaries in the spatial dimensions are communicated, thus incurring the same communication volume cost as standard tiling.

2.6 Code Generation

In this section, we discuss the generation of the code for the iteration space with the overlapped and split tiles. We describe the derivation of the parameters necessary to utilize the code generation framework described by Ancourt and Irigoin [7].

Each tile in the tiled iteration space is identified by a tile origin. The execution of the tiled iteration space is defined as the traversal of the tiles in terms of their origins, together with the execution of the iterations mapped to each tile as it is traversed.

The origin of the tiled iteration space defined to be the origin of the original iteration space. Given the origin, all the tile origins can be enumerated as linear combinations of the shift vectors. The tile size is defined as the distances between the tile origins along the shift vector, and is embedded in the specification of the shift vector itself.

The matrix of shift vectors specifies the traversal order of the tile origins. The shift vectors are ordered to enable an outer loop along the direction b_i so that there is parallelism-inner synchronization-outer.

Given the tile origin x_0, defined equivalently in terms of the shift vectors or as iteration points in the original iteration space, each of the hyperplanes bounding the tiles can be identified by a point in it. For hyperplanes h_i along which no overlap is identified as necessary, the iteration points x in the iteration space that form this tile satisfy the following conditions:

$$h_i.x \geq h_i.x_0 \wedge h_i.x < h_i.(x_0 + s_i)$$

Note that x_0 is a vertex on all the non-overlapped hyperplanes that form the back face of the tile. $x_0 + s_i$ is a point on the front face of the tile for all hyperplanes h_i. Since overlapping does not change the front face, this is also true for hyperplanes that utilize overlap.

When an overlapped hyperplane is identified along a dimension, we replace the back face of the original hyperplane h_i by an overlapped hyperplane h'_i. Since h'_i is constructed from h_i by only shifting it along the other hyperplanes, the point $x_0 + \sum_{j \neq i} s_j$ is a valid point on it irrespective of the choice of h'_i. Thus the boundary conditions for the tile for these hyperplanes is given by:

$$h_i.x \geq h_i.(x_0 + \sum_{j \neq i} s_j) \wedge h_i.x < h_i.(x_0 + s_i)$$

Given the tile origins and their traversals, and the shape of the overlapped tile, the code generation procedure of Ancourt and Irigoin [7] can be used to generate code. The generated code would have n outer tile space loops, each corresponding to a tiling hyperplane, and inner loops enumerating all iterations belonging to a tile. Let us assume that k of the n hyperplanes have been identified for overlapped tiling. Overlapped tiling enables concurrent start along a hyperplane by eliminating any inter-tile dependence along that hyperplane. Hence, the tile space loops corresponding to the remaining $n-k$ hyperplanes carry all inter-tile dependences, and can be run sequentially as the outer loops, and the k tile space loops corresponding to overlapped tiling hyperplanes can all be run in parallel by mapping to a k-dimensional or lower dimensional processor space.

The traversal of tile origins for split tiling is the same as that for standard tiling. The intra-tile code is generated for the various tile components by scanning the polytopes derived by specifying the appropriate hyperplane instances that bound the tile component, as defined earlier. The appropriate hyperplane boundaries between sub-tiles define the data to be communicated between processors for the communication phases, as discussed earlier.

2.7 Experimental Evaluation

Both the proposed tiling schemes—overlapped tiling and split tiling—enable load-balanced tiled execution of stencil codes that inherently satisfy the concurrent-start criterion. The degree of exploited concurrency is the same with both schemes; they differ in the computation/communication overheads relative to standard tiling. With overlapped tiling, there is a small amount of computational overhead and also a small increase in the total communication volume. Split tiling requires no additional redundant computations and requires exactly the same total communication volume as standard tiling, but requires additional messages, i.e., incurs a higher message-startup-cost overhead.

Below, we report experimental results comparing overlapped/split tiling with standard (pipelined) tiling for the one-dimensional Jacobi code. The experiments were conducted on a cluster consisting of 32 compute nodes each of which is a 2.8 GHz dual-processor Opteron 254 (single core) with 4GB of RAM and 1MB L2 cache, running Linux kernel 2.6.9. We used one processor per node in our experiments. The code was compiled using the Intel C Compiler with -O3 optimization flag.

The iteration space of one-dimensional Jacobi has a space dimension and a time dimension. Two versions of pipelined schedule were implemented: (i) one in which the processor space was mapped along the time dimension and time along the space, and (ii) the other one in which the processors were distributed in a block-cyclic fashion to execute tiles along time dimension.

First we conducted experiments to determine the optimal time tile size and space tile size for the two pipelined schedules. The experiments were conducted for 1000 time steps on 32 processors for a total problem size of 64000 elements. The execution times are shown in Figures 2.9 and 2.10. The number of communication startups decreases with an increase in the spatial tile size. This typically results in a decrease in the execution time with an increase in the space tile size. But for larger space tile sizes, the pipeline startup costs increase thus dominating and increasing the execution time. Increase in the time tile size reduces the number of time tiles and hence the number of synchronizations. But larger time tile sizes as in the case of larger space tile sizes increase the pipeline startup costs. Hence an increase in the time tile size decreases the execution time until the pipeline startup costs begin to dominate. The execution times for both the pipelined schedules, as inferred from the experiments, are minimum for a time tile size of 16 and space tile size of 1000. Hence a time tile size of 16 and space tile size of 1000 were used for subsequent evaluation of the schemes.

For overlapped and split tiling, the space tile size is fixed to be $N/nproc$, where N is the space dimension size and $nproc$ is the number of processors used for parallel execution. The time tile size is chosen to be 16 to match the choice for the pipelined schedules.

Given these choices of space and time tile sizes, the performance of the four schemes for various problem sizes is shown in Figure 2.11. The split and overlapped tiling schemes result in a linear increase in execution time with problem size, unlike the pipelined tiling solutions. The improvement in execution time achieved by split and overlapped tiling schemes with increase in problem size is due to the better exploitation of data locality. In addition, unlike the pipelined schedules, the communication cost is independent of the problem size.

The improved scalability of the overlapped and split tiling schemes, due to an absence of the pipeline startup cost, is shown in Figure 2.12. The problem size was fixed at 20000 elements per processor. The number of processors was varied to measure the weak scaling capability of the various schemes. A straight line parallel to the x-axis corresponds to linear scaling. The split tiling solution performs best, followed by the overlapped tiling solution. The pipelined schedules suffer from performance degradation with increase in the number of processors.

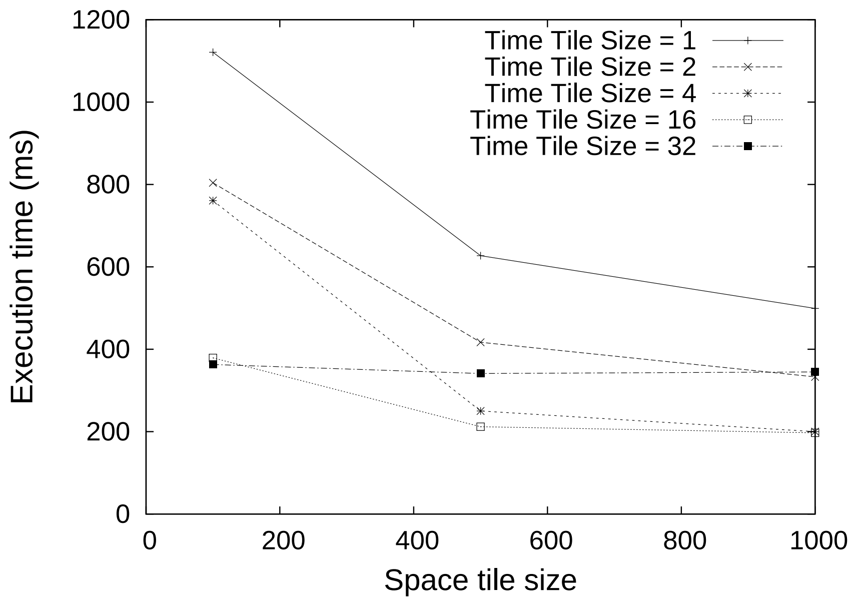


Figure 2.9: Optimal space and time tile size for pipelined schedule 1

2.8 Related Work

Several recent works have presented manual optimizations and experimental studies on stencil computations [54, 53, 39]. Iteration space tiling [52, 119] is a method of aggregating a number of loop iterations into *tiles* where the tiles execute atomically; communication (or synchronization) with other processors takes place before or after the tile but not during the execution of the iterations of a tile. Several works have used tiling for exploiting data locality [4, 3, 118, 106, 29]. Others have addressed the selection of tile shape and size to minimize overall execution time [34, 92, 9, 95, 49, 14]. The size of tiles has an impact on the amount of parallelism and communication: smaller tiles increase parallelism by reducing pipelined startup cost, while larger tiles reduce frequency of communication among processors. This has been studied by a number of researchers [9, 95, 49, 50, 33, 51]. Griebl

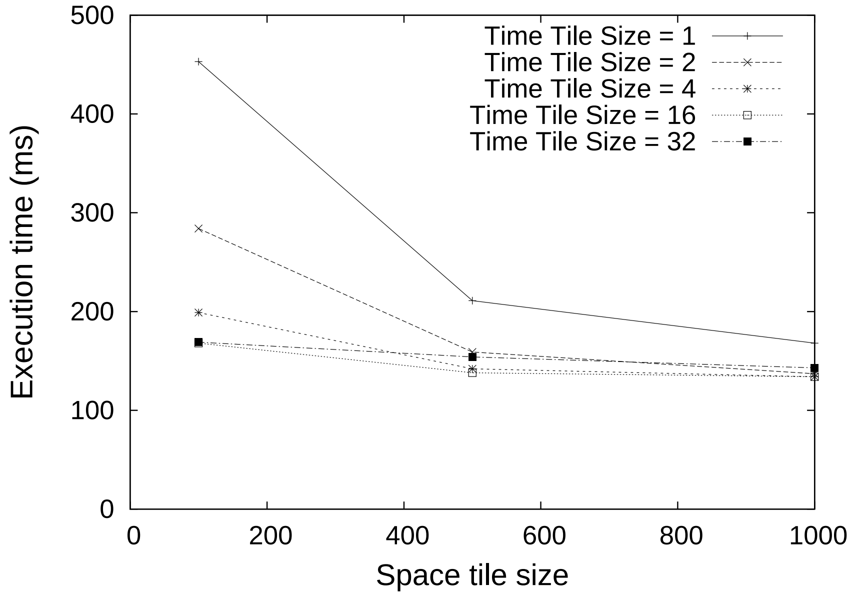


Figure 2.10: Optimal space and time tile size for pipelined schedule 2

[42, 43] presents an integrated framework for optimizing data locality and parallelism in the use of tiling; however, pipelining issues are not considered.

Li and Song [77] present techniques to optimize stencil codes through loop skewing and array padding. Strout [] present techniques for data and computation reordering for sparse matrix computations. The optimizations include time tiling for relaxation codes, with tile shapes similar to that derived by our overlapped tiling approach.

The Omega toolkit [59] provides support to compute the exact transitive dependences of tuple relations when possible, if not resort to computing a lower bound. Kelly et al. [61] present an approach to compute transitive closure of parameterized tuple relations and present its applications, including the determination of transitive dependences. Pugh and Wonnacott [91] present techniques to compute both upper and lower bounds of transitive closures. These techniques can be employed to determine the transitive dependences that

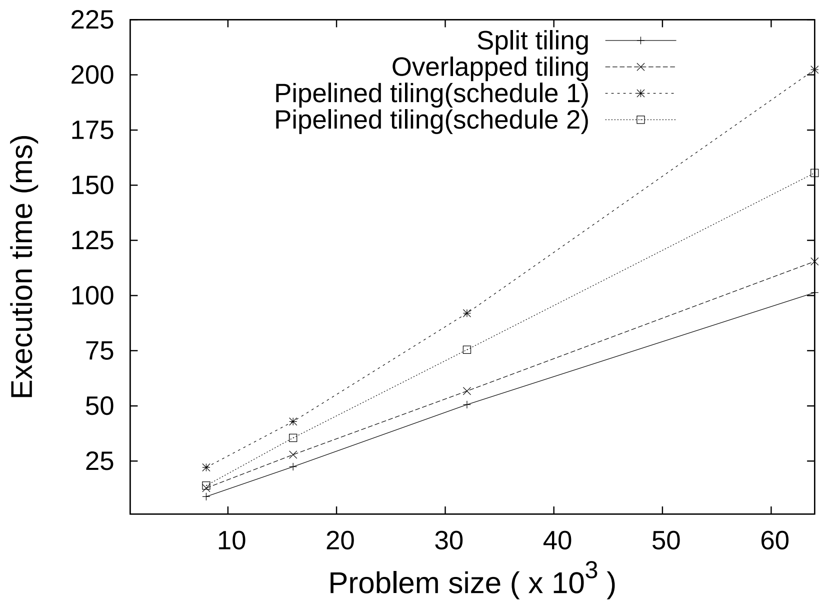


Figure 2.11: One-dimensional Jacobi execution time, varying problem size

we are interested in. Wonaccott [121] discusses time skewing to optimize locality of stencil computations. In the parallel context [120], the work presents an approach similar to split tiling discussed here. It does not consider overlapped tiling, or present a characterization of when it is beneficial.

Sawdey and O'Keefe [99] describe TOPAZ the tool that explores the replicated computation of boundary values in the context of SPMD execution of stencil codes, in which the user marks regions of code to be replicated; the tool then analyzes and generates the correct code. This approach helps with reducing communication costs and improving load balance. Adve et al. [1] describe computation partitioning strategies used in the dHPF compiler that exploit replicated computation using the `LOCALIZE` directive that is available in dHPF. Both these approaches rely on user-specification of replicated computation, unlike our approach to automatic parallelization.

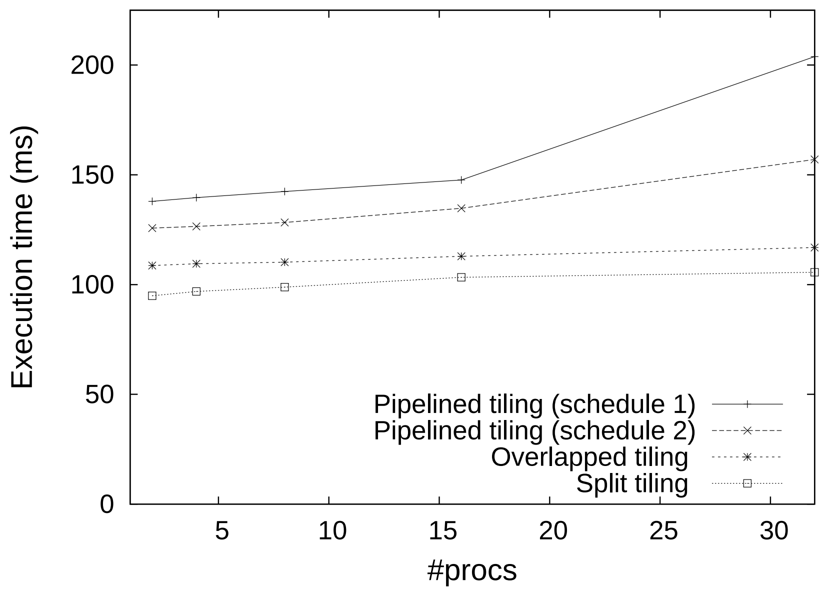


Figure 2.12: One-dimensional Jacobi execution time, varying #procs

2.9 Conclusions

Iteration space tiling has received considerable attention motivated by optimizing for data locality as well as by exploiting parallelism for nested loops. The choice of the shape of iteration space tiles may result in inter-tile dependences that inhibit concurrent execution of tiles on different processors, leading to a pipelined start overhead. This chapter has addressed the issue of enhancing concurrency with tiled execution of loop computations with constant dependences. Two approaches, namely *overlapped tiling* and *split tiling* were presented, that enabled the removal of inter-tile dependences, thereby enabling additional concurrency. These techniques expose the trade-off between the communication and computation costs, and the parallelism in the program. Experimental results demonstrated the effectiveness of the proposed schemes on stencil codes.

CHAPTER 3

DATA LAYOUT TRANSFORMATION FOR DISK RESIDENT ARRAYS

3.1 Introduction

Many scientific and engineering applications need to operate on data sets that are too large to fit in the physical memory of a machine. Such data is stored in disk and brought into physical memory for processing as needed. The data is then said to reside *out-of-core* and the program is referred to as an *out-of-core program*.

The bandwidth available to access data in secondary storage is much smaller than from main memory, and this discrepancy is only exacerbated by current technology trends . This necessitates minimization of disk access while maximizing reuse of data already in memory. In addition, the extremely large seek time relative to the per-word transfer time for disk access dictates that I/O be done using contiguous blocks of disk resident data. These concerns can require careful reexamination of an in-memory algorithm to tailor it to the characteristics of secondary storage.

An approach to solving this problem exploits the operating system's virtual memory. The user addresses data in an address space often larger than the physical memory of the machine. The operating system implicitly moves the data from secondary storage to physical memory when it is accessed by the user and replaces other unused data to free up physical memory as needed. Any modified data is written back to disk before being replaced. Since the data movement is done in units of an operating system page, improved disk I/O bandwidth is achieved. While providing a simple abstraction, this approach suffers from several drawbacks. First, the generic page replacement policies in kernels do not exploit the specialized data access patterns exhibited by scientific applications [17]. Second, the virtual memory supported by 32-bit operating systems is still too small compared to the disk space available even in a single hard disk drive. Third, efficient extensions of virtual memory for parallel systems are not available.

An alternative approach commonly employed acknowledges secondary storage as another level of the addressable memory hierarchy and explicitly moves data between main memory and secondary storage. Higher-level abstractions are provided to enable simplified yet efficient data movement where possible.

We focus on disk I/O support to enable simplified yet efficient abstractions to access multi-dimensional arrays stored on disk. I/O libraries like PANDA [102, 109] and DRA [38] use a blocked representation for the disk-based multidimensional arrays to optimize performance of collective I/O operations between arrays located on disk and in the distributed main memory of parallel computers [22].

Thus a disk-based multidimensional array is partitioned into a number of multidimensional blocks or “bricks” and the elements within a brick are linearized using some dimension order. Unlike the dimension-ordered representation typically employed to represent in-memory multidimensional arrays, the bricked representation permits efficient contiguous access as long as the accessed regions mostly contain full bricks.

However, in some programs, the access patterns to some disk-based multidimensional arrays in two successive phases (or the access pattern of the producer and the consumer) are so different that no choice of brick shape will allow for efficient access in both the phases. An example is the out-of-core two-dimensional Fast Fourier Transform (FFT), where the array is accessed by columns in one phase and by rows in the other. The multi-dimensional FFT [8, 10] can be implemented as a series of one-dimensional FFTs, one along each dimension. As another example, consider image data in three and four (including time) dimensions. The production of data from scanning occurs plane by plane. However, examination of the time evolution of a three-dimensional block of data requires a very different access pattern than that by which the data was generated. In isosurface construction in three and four dimensions, the data is typically produced in a row-major format by scanning or simulation. The amount of memory available determines the amount of data generated between writes to disk, and hence limits the blocking possible [58]. In such scenarios the performance of computations operating on the stored data might be greatly improved by transforming the data into a different blocked form to match the application's access pattern.

In this chapter, we present efficient data layout transformation algorithms to transform the blocking of multi-dimensional arrays. We first develop an out-of-core matrix transposition algorithm that takes into account the I/O characteristics of the target system. We then present a novel algorithm to solve the out-of-core matrix reblocking problem for multi-dimensional matrices of arbitrary sizes.

System	Configuration			
	Processor	Memory (MB)	Linux	Compiler
ia64-osc	Dual Itanium-2 (900 MHz)	4096	2.4.18	gcc 2.96
amd-osc	Dual Athlon MP (1.533 GHz)	2048	2.4.20	pgcc 4.0-2

Table 3.1: Configuration of systems used for I/O characterization

3.2 Disk I/O Characterization

Out-of-core algorithms on multi-dimensional arrays, such as out-of-core matrix transposition, involve reading and writing blocks of data at different strides. To understand the variation in performance of the algorithm with respect to these parameters, we studied the variation of read and write times with changes in size and stride of I/O on two clusters at the Ohio Supercomputer Center (OSC) [87]. The configuration of each compute node in these clusters is shown in Table 3.1. Figure 3.1 and Figure 3.2 show the strided read and write times respectively on amd-osc. Figure 3.3 and Figure 3.4 show the strided read and write times respectively on ia64-osc.

On both systems we observe that beyond a particular block size the stride does not affect the per-byte transfer cost and approximates to the cost of sequential I/O. More importantly, the incremental improvement obtained in the I/O time by increasing the block size decreases and is very small beyond a particular block size. We expect this observation to hold across a wide variety of systems. These block sizes, above which the per-byte read and write times are not affected by the stride of access, will henceforth be referred to as the *read* and *write thresholds*, respectively. These parameters vary depending on the system under consideration and the per-byte read and write costs can saturate at different

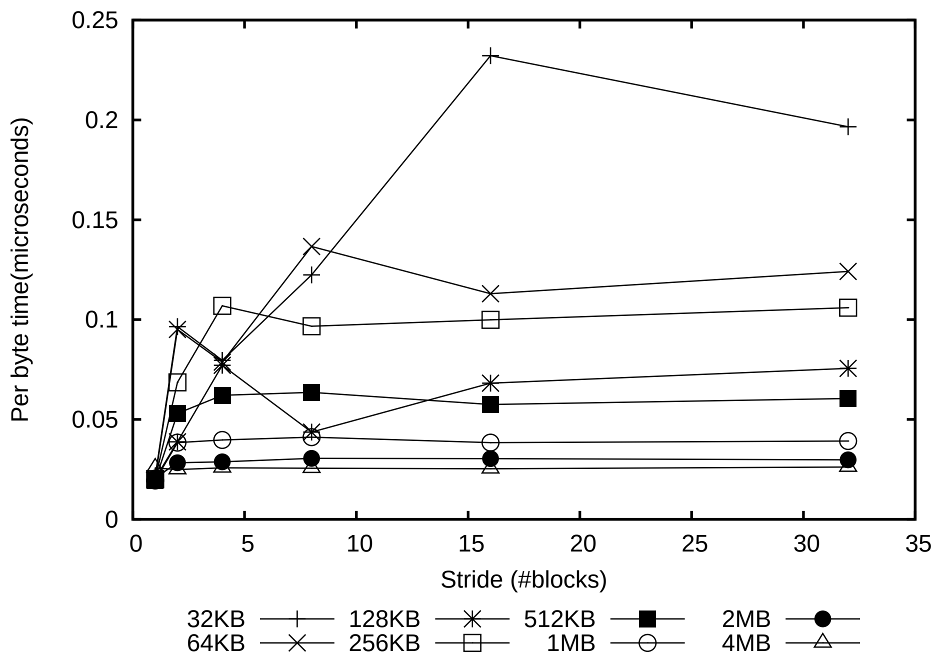


Figure 3.1: Strided read times on amd-osc

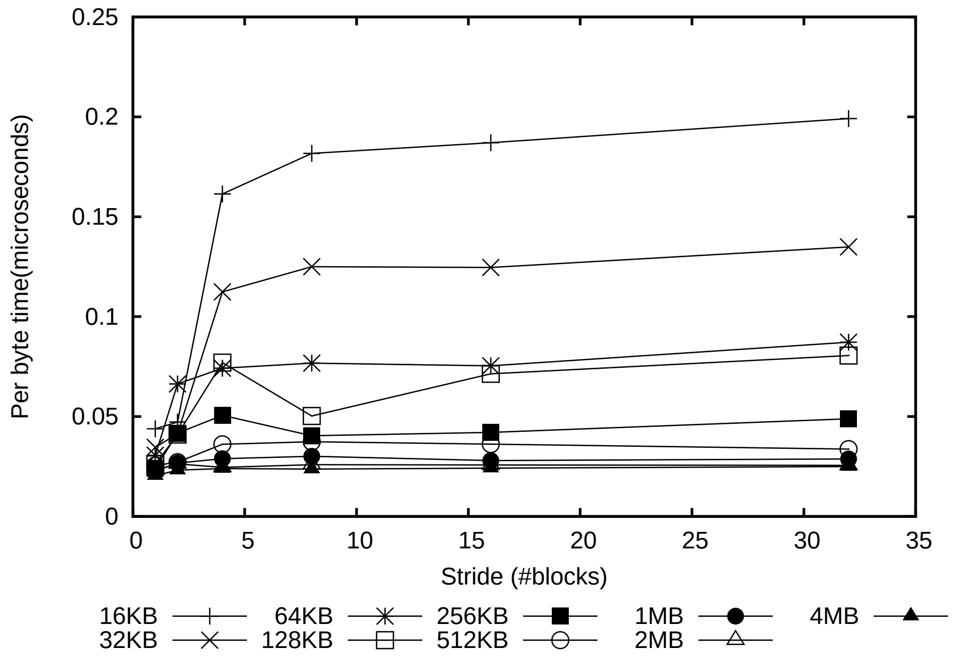


Figure 3.2: Strided write times on amd-osc

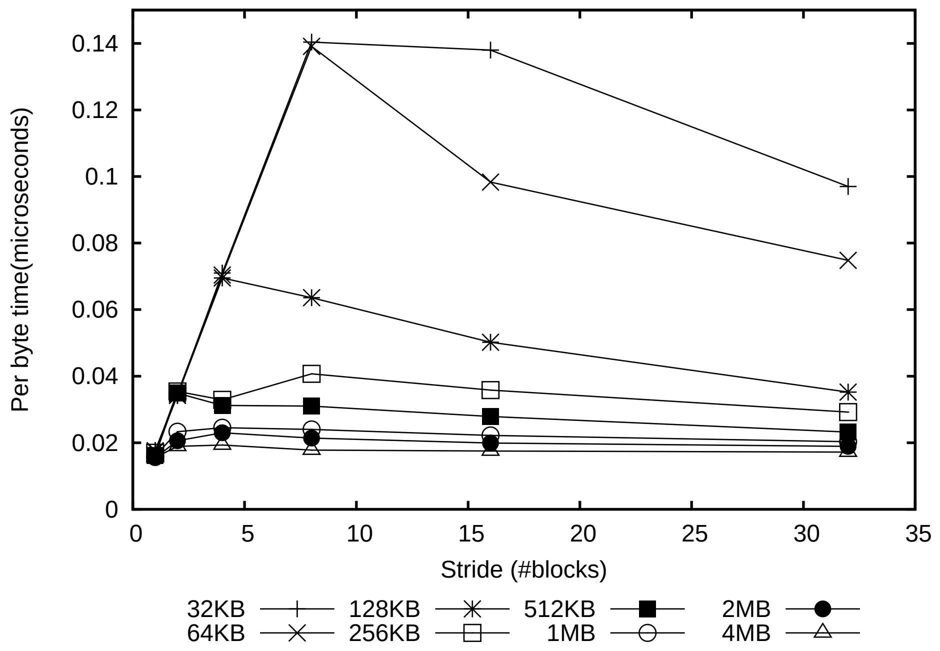


Figure 3.3: Strided read times on ia64-osc

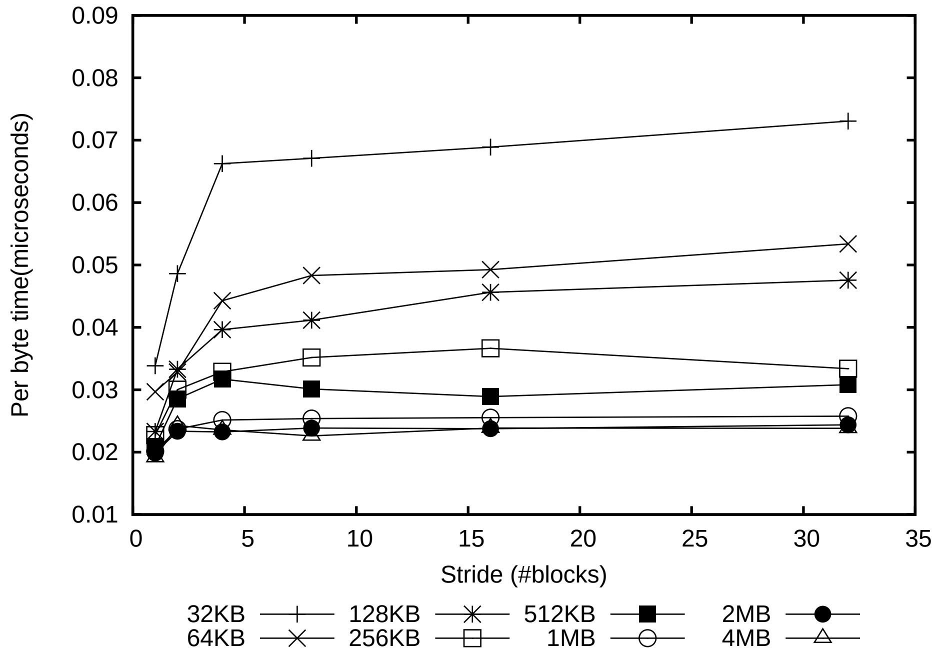


Figure 3.4: Strided write times on ia64-osc

block sizes. The read and write thresholds on amd-osc are 2MB and 1MB, respectively. On ia64-osc, they are both 1MB.

An out-of-core algorithm needs to perform I/O on sufficiently large block sizes for good performance. On the other hand, a smaller block size provides greater flexibility in accessing the data and can improve performance of the algorithm. An out-of-core algorithm may not be optimal if it chooses the largest possible I/O block size when I/O on a much smaller block can be performed efficiently. It is possible for an algorithm with more I/O operations to be faster than another algorithm with fewer I/O operations. In the case of out-of-core matrix transposition, if the thresholds are smaller than N, the size of the matrix, fractions of a row can be read and written with little additional penalty, irrespective of the stride of access. In the extreme case, if each element is large enough to be read/written individually, a simple single-pass element-wise transposition would be the most efficient.

3.3 Out-of-core Matrix Transposition

3.3.1 Problem Definition

Consider an $N \times N$ matrix that is stored in disk in row-major order. The system has main memory, which can hold M elements, where $M < N^2$, $M = O(N)$. Each element of the matrix is too small to be read from and written to disk efficiently. The problem is to transpose the matrix stored in disk, when only a portion of the matrix can be brought into memory at any time. Matrix transpose is a key operation in various scientific applications. For example, the two-dimensional Fourier transform [8, 10] can be implemented as a one-dimensional Fourier transform along the rows, followed by a one-dimensional Fourier transform along the columns. For a matrix stored in disk in row-major order that is too large to fit in memory, the most effective mechanism is to transpose the matrix before the second pass.

This problem has been widely studied in the literature. A simple in-place element-wise approach to transpose the matrix is prohibitively expensive as long as each element is not large enough to be read (written) from (to) disk efficiently. The block transposition algorithm transposes the array in a single pass in $O(N^{3/2})$ I/O operations, where a pass is defined as accessing each element from disk exactly once. An in-place transposition algorithm requiring $O(N \log N)$ disk accesses was proposed by Eklundh [37]. This algorithm requires at least two rows to fit in memory. Extensions to the algorithm for rectangular matrices were later developed [6, 93, 113]. Kaushik et al. [57] improved upon these algorithms by reducing the number of read operations. Suh and Prasanna [107] reduced the in-memory in-place permutation time by using collect buffers, instead of in-memory permutation, in addition to reducing the number of I/O operations. All these studies use the number of I/O operations as the primary optimization metric.

Although the execution time of the solution provided has been improved by all these efforts, the total execution time has not been used as the primary metric for optimization. A reduction in the number of I/O operations, in most cases, translates to larger sizes of I/O blocks. The importance given to reducing the number of I/O operations is due to the fact that the disk access time, comprising disk seek time plus latency, is very large (on the order of several milliseconds) compared to the per-byte transfer time (on the order of microseconds or less). If the I/O blocks read/written are relatively small, the total number of I/O operations is indeed a suitable optimization metric. However, when the I/O blocks get large, the data transfer time becomes significant and can dominate the total access time. In such a situation smaller block sizes can be read/written without any additional I/O cost.

But this might reduce the number of passes involved, thus improving performance. Since previously proposed algorithms for out-of-core transposition have focused on reducing the number of I/O operations, they can become sub-optimal when large block transfers are involved.

All the algorithms in the literature determine the fundamental unit of I/O based on the size of the matrix, i.e., they are data-centric. The basic unit of I/O operation in these algorithms is one row of the matrix or a multiple thereof. They do not adapt to the I/O characteristics of the system. In contrast, the approach proposed here takes into account the empirically determined I/O characteristics of the disk and file system. The parameters of the algorithm, including the basic unit of I/O and the estimate of the execution time of the algorithm, are determined based on the empirically measured I/O characteristics.

3.3.2 Matrix Transposition Algorithms

In this section, we discuss some of the out-of-core matrix transposition algorithms from the literature. The pseudo-code for the algorithms is given with focus on the I/O operations performed in each algorithm. These algorithms are formalized in the next section.

Consider a square matrix of dimension $N = 2^n$. Let the number of elements that can be brought into memory at any time be $M = 2^m$. The memory can hold $B = 2^b$ rows, say, of the input matrix, i.e., $B = M/N$. Each algorithm runs in a certain number of passes. Each pass involves reading the entire array from disk and writing it back. In each pass, the algorithm goes through a sequence of steps, each of which involves three phases–reading data into memory, permuting the in-memory data and writing it back to disk. All algorithms proceed as a sequence of steps in each pass. A step is defined as the operations performed between reading a portion of data into memory and writing it back to disk, including the read and

```
1: for $i = 0$ to $N/\sqrt{M} - 1$ do
2:     for $j = 0$ to $N/\sqrt{M} - 1$ do
3:         Read data range $[i * \sqrt{M} : (i+1) * \sqrt{M} - 1][j * \sqrt{M} : (j+1) * \sqrt{M} - 1]$
4:         Transpose in memory
5:         Write data range $[j * \sqrt{M} : (j+1) * \sqrt{M} - 1][i * \sqrt{M} : (i+1) * \sqrt{M} - 1]$
```

Algorithm 3.1: Block transposition algorithm

write operations. All algorithms in the literature work on disjoint ranges of data in each step. Note that the algorithms discussed can be employed to transpose matrices whose size is not a power of 2. We capture the basic idea of each algorithm and provide a formulation for the out-of-core matrix transposition problem. This formulation is used to arrive at a better algorithm.

The block-transposition algorithm is a single-pass algorithm for matrix transposition. The algorithm blocks the input matrix into smaller matrices and recursively transposes the embedded matrices. Algorithm 3.1 presents the block-transposition algorithm. Each step of the algorithm involves $\sqrt{M}$ read and write operations, each of $\sqrt{M}$ elements. The algorithm reads and writes at different locations in the matrix in any given step, thus requiring the destination array to be different from the source array, i.e., the algorithm is out-of-place. Note that the algorithm can be implemented as an in-place algorithm at the expense of increased memory usage, similar to in-place in-memory matrix transposition algorithms [69, 15, 20].

Eklundh's algorithm [37] does the transposition in-place in n/b passes. This algorithm, shown as Algorithm 3.2, requires that $n \bmod b = 0$. Each step of the algorithm involves M/N read and write operations, each involving N elements.

Kaushik et al. [57], shown as Algorithm 3.3, improve upon Eklundh's algorithm by combining the reads. It is an out-of-place algorithm. In each step of the algorithm, one

```
1: for $i = 0$ to $n/b - 1$ do
2:     for $j = 0$ to $N^2/M - 1$ do
3:         Read $(M/N)$ rows starting with $(\lfloor (j/B^i) \rfloor * B^{i+1} + j\%B^i)$-th row at a stride of $B^i$ rows
4:         Permute in memory
5:         Write to the rows from which the data was read
```

Algorithm 3.2: Eklundh's algorithm

```
1: for $i = 0$ to $t - 1$ do
2:     for $j = 0$ to $N^2/M - 1$ do
3:         Read $(M/N)$ contiguous rows starting at $(j * M/N)$-th row
4:         Permute in memory
5:         for $k = 0$ to $M/s_i - 1$ do
6:             Write $s_i$ rows starting at $(k * s_i)$-th row in memory to the array in disk starting at
                   the $(j * (M/N)/s_i + k)$-th row at stride $N/s_i$ rows
```

Algorithm 3.3: Kaushik et al.'s algorithm

read of M elements and M/N writes, each of N elements, are performed. In the most general case, N is factorized into $s_0 * \ldots * s_{t-1}$ such that for any s_i, s_i rows fit in memory. The algorithm runs in t passes. Kaushik et al. provide a solution when only one row fits in memory, which cannot be handled by Eklundh's algorithm. They also provide a mechanism to use the maximum available memory.

Suh and Prasanna's algorithm improves further upon Kaushik's algorithm in two ways. It reduces the in-memory permutation time by replacing in-place permutation by a series of collect operations, in which the data to be written is collected into a buffer. The algorithm also reduces the number of I/O operations by 'chunking' the writes. The writes that would have been done at different offsets are done contiguously. This increases the write size and reduces the number of writes. Each write operation in the $i-$th pass writes $z_i * N$ elements instead of N elements as written by Kaushik et al. In the subsequent pass, the

```
1: for $i = 0$ to $t - 1$ do
2:     for $j = 0$ to $N^2/M - 1$ do
3:         Collect $(M/N)$ rows that have been separated by $z_{i-1}$ rows in the previous pass   ▷ Might involve multiple reads
4:         Permute in memory
5:         Write the permuted data to disk with $z_i$ rows in each I/O operation   ▷ Might involve multiple writes
```

Algorithm 3.4: Suh and Prasanna's algorithm

data that should have been written contiguously is 'collected' by performing a sequence of reads. Thus the number of reads is increased from one in Kaushik et al. This mechanism balances the number of reads and writes. The optimal value for z_i was determined to be $\sqrt{s_i}$, at which point the number of writes equals the number of reads and the total number of I/O operations is minimum. In the algorithms discussed so far, each element is read into memory exactly once in each pass. On the other hand, each pass i in this algorithm performs redundant reads to first collect the rows, that have been separated by z_{i-1} rows by the previous write, into memory and then performs the permutation. This increases the memory usage and potentially the total I/O cost if the memory available is not sufficient to retain all the read data in memory before the permutation can be performed.

3.3.3 Formulation of Transposition Algorithms

In this section, the matrix transposition algorithms are formulated using the matrix vector product notation described by Edelman et al. [36]. Transposition of a matrix can be viewed as an interchange of the indices of the matrix. This is a particular instance of the more general class of index transformation algorithms.

Each element of the array has a linear address vector obtained by concatenating the column index to the row index, both indices being represented as bit vectors. Transposition corresponds to a transformation of this linear address vector and can be represented by a transformation matrix.

The identity of the transformation is I_{2n}. Matrix transposition is defined as the transformation of the address vector i

$$i \rightarrow Ti$$

where T is the transformation matrix $\begin{pmatrix} 0 & I_n \\ I_n & 0 \end{pmatrix}$.

We use the following notation in the discussion. Given two matrices A and B

$$A \oplus B = \begin{pmatrix} A & 0 \\ 0 & B \end{pmatrix} \tag{3.1}$$

$$L(A,B) = \begin{pmatrix} 0 & B \\ A & 0 \end{pmatrix} \tag{3.2}$$

$L(I_n, I_n)$ is the desired transformation. Since the transformed cannot be effected efficiently for out-of-core matrices except with very large element sizes, $L(I_n, I_n)$ is factorized into a number of transformation matrices such that the transformation effected by each of the matrices can be done efficiently with the memory available. The following discussion provides the matrix vector formulation of various out-of-core matrix transposition algorithms discussed in the previous section.

Any out-of-core matrix transposition algorithm consists of three phases–read, permute and write. Each phase is modeled by a transformation matrix. These phases are repeated on disjoint sets of data in the different steps of each pass. The algorithm might involve many passes, each operating on the entire array. Thus, out-of-core matrix transformation algorithms are of the form

$$L(I_n, I_n) = \prod_{i=t-1}^{i=0} W_i P_i R_i$$

where W_i is the transformation matrix corresponding to write, R_i is the transformation matrix corresponding to read and P_i corresponds to in-memory permutation for the ith pass. t is the number of passes. The algorithms under this formulation read some data, permute it in memory, and write the data to disk before reading data for the next step in the same pass. Each algorithm is defined by the parameters t, W_i, P_i, and R_i.

Some restrictions apply to the possible values of W_i, P_i and R_i. Each transformation matrix must correspond to a transformation of the given out-of-core matrix that can be efficiently done with the memory available. Thus, each step of the algorithm can operate on at most M elements. In particular, W_i, P_i, and R_i must be expressed as

$$\begin{aligned} R_i &= A_{2*n-r} \oplus I_r \quad r \leq m \\ P_i &= I_{2*n-m} \oplus B_m \\ W_i &= C_{2*n-w} \oplus I_w \quad w \leq m \end{aligned}$$

The restriction of R_i shows that the unit of read must be at least $R(= 2^r)$ elements. The transformation in the read operation as modeled by A determines the pattern of reads. Similar restrictions apply for write operations. The restriction on P_i shows that in-memory permutation can transform only address elements corresponding to the data elements in memory. Given these parameters, an out-of-core transposition algorithm can be implemented as shown in Algorithm 3.5.

3.3.4 Performance Analysis

In this section, we analyze the performance of various algorithms based on their formulation. The parameters R_i, P_i, and W_i of each algorithm are determined and are used to analyze the performance of the algorithm.

```
1: for $i = 0$ to $t-1$ do
2:     for $j = 0$ to $N^2/M - 1$ do
3:         Read $M$ elements at address $R_i^{-1}(j)$        ▷ Might involve multiple reads
4:         Permute in memory according to $P_i$
5:         Write $M$ elements at address $W_i(j)$        ▷ Might involve multiple writes
```

Algorithm 3.5: Generic transposition algorithm

The running time of an algorithm depends on the read, write, and in-memory permutation times. The I/O time depends on the size and stride of I/O. r and w determine the read and write block size, respectively, and hence are important parameters. In addition, the stride of access plays an important role, as demonstrated by the I/O characteristics. The strides of reads and writes are determined by the A and C sub-matrices, respectively. For some algorithms it might be possible to rewrite A (C) as $D \oplus I_k$, for some matrix D. In such cases the read (write) sizes can be larger than $2^r(2^w)$ elements.

Block Transposition

The $2n$ bits are partitioned into four components

$$I_{2n} = I_{RH} \oplus I_{RL} \oplus I_{CH} \oplus I_{CL}$$

such that $RL + CL = m$. The parameters are

$$\begin{aligned}
t &= 1 \\
R_i &= (I_{RH} \oplus L(I_{RL}, I_{CH}) \oplus I_{CL}) \\
P_i &= (I_{RH+CH} \oplus L(I_{RL}, I_{CL})) \\
W_i &= (L(I_{RH}, I_{CH} \oplus I_{CL}) \oplus I_{RL})
\end{aligned}$$

It is a single pass algorithm. The algorithm reads 2^{RL} elements and writes 2^{CL} elements in one I/O operation. Generally the components are chosen such that $RL = CL$ producing square blocks. The I/O size is typically $O(\sqrt{M})$ elements. Even for large memory sizes, this would fall short of reaching the threshold leading to high I/O cost, making this a very inefficient algorithm. Note that this algorithm is not inefficient due to the large number of I/O operations involved ($O(N^{3/2})$), but because of the small I/O size. For systems with memory large enough to make $O(\sqrt{M})$ larger than the threshold, this algorithm is optimal.

But a more effective way of choosing RL and CL would be to minimize the total I/O cost. Thus the problem becomes

$$\begin{aligned} RL + CL &= m \\ minimize &: cost(read) + cost(write) \end{aligned}$$

A cost model for read and write can be derived from the I/O characteristics of the system. These cost equations can be used to arrive at the best parameters for the algorithm.

Eklundh's Algorithm

Eklundh's algorithm [37] has the following formulation:

$$\begin{aligned}
b &= m-n \\
I_{2n} &= I_{n-b} \oplus I_b \oplus I_n \\
t &= n/b \\
R_i &= (I_{n-(i+1)*b} \oplus L(I_b, I_{i*b}) \oplus I_n) \\
P_i &= (I_{n-b} \oplus L(I_b, L(I_{n-(i+1)*b}, I_b)) \oplus I_{i*b}) \\
W_i &= (I_{n-(i+1)*b} \oplus L(I_{i*b}, I_b) \oplus I_n)
\end{aligned}$$

R_i^{-1} (=R_i^T as R_i is a permutation matrix) and W_i are identical, indicating that the algorithm can be executed in-place. Each phase (read, write, and permute) of the algorithm depends on the pass in which it occurs. The algorithm reads and writes N elements in each I/O operation, independent of the I/O characteristic of the underlying system. Hence the algorithm might perform well on some machines and poorly on others. In addition, unless the matrix size (N^2) is of the order of terabytes, N is lower than the read threshold for the systems analyzed in Section 3.2.

Kaushik’s Algorithm

Kaushik’s algorithm discussed in Section 3.3.2 can be formulated in the following manner, given $s_i = b$, $0 \leq i < t$.

$$\begin{aligned}
b &= m - n \\
I_{2n} &= I_{n-b} \oplus I_b \oplus I_n \\
t &= n/b \\
R_i &= (I_{2n}) \\
P_i &= (I_{n-b} \oplus L(I_n, I_b)) \\
W_i &= (L(I_{n-b}, I_b) \oplus I_n)
\end{aligned}$$

This is an out-of-place algorithm involving t identical passes. The algorithm reads M elements in one I/O operation, thus comfortably achieving the read threshold. Each write involves N elements. This algorithm improves on Eklundh’s by reducing the read costs by performing sequential reads of large size. This algorithm does not take advantage of the I/O characteristics of the system, by writing smaller block sizes than a row, if little additional cost is incurred. The in-memory permutation phase in every pass involves element-wise permutation, unlike Eklundh’s algorithm which moves larger blocks with each pass. This could increase the in-memory permutation cost as compared to Eklundh’s algorithm.

Suh and Prasanna’s Algorithm

This algorithm does not fit into the formulation discussed as it might involve redundant reads. This algorithm improves upon Kaushik’s by reducing the number of I/O operations and increasing the I/O size of writes. Instead of writing one row at a time to enable contiguous read in the next pass, the rows transformed by each step are chunked and written back to disk. In the next pass, each read phase collects the rows to be transformed in the current step from the chunks. Each row in a chunk is transformed in a different step in the next pass. If a chunk brought into memory can be retained until the steps involving all its rows can be processed, this algorithm improves upon Kaushik's algorithm by balancing the number of read and write operations and sizes. However, in attempting to balance the number of read and write operations the memory available is not taken into account. When the chunks cannot be retained until they are fully processed, redundant I/O is incurred. Hence the performance of the algorithm can vary dramatically depending on the parameters of the problem. The algorithm usually benefits from an increase in memory, since an increase in memory reduces the redundant data movement incurred.

3.3.5 Sequential Out-of-Core Matrix Transposition

Our algorithm tries to minimize the I/O time involved by choosing the parameters appropriately. The observation that an increase in I/O size beyond the threshold does not influence the performance of the algorithm is exploited. There is a trade-off between the I/O size and the number of passes the algorithm requires. The smaller the I/O size, the more the algorithm approaches the block-transposition algorithm and runs in a smaller number of passes. However, reducing the I/O size below the threshold increases the I/O time above the minimum possible.

The formulations of all algorithms discussed so far require two parameters – m and n – to derive a concrete list of operations. Our algorithm requires two additional parameters; namely, the read block size (2^r) and the write block size(2^w). These are chosen to be close to the threshold, with the exact value depending on the number of passes required for the given block sizes. Smaller block sizes incur more I/O time but might potentially reduce the number of passes, thus significantly reducing the total time. The most common scenario in which an I/O block size smaller than the threshold is chosen is when such a choice reduces the number of passes and offsets the additional cost incurred due to the smaller I/O size.

The number of rows to be transformed in each pass is determined as the maximum possible. The chunking factor, the factor which determines the extent of chunking similar to that in Suh's algorithm, is chosen so that no redundant reads are incurred. This provides the benefits of chunking, such as increasing the I/O size, without increasing the total I/O time.

In other algorithms the basic unit of I/O is a row. The I/O transformation matrices are of the form $A \oplus I_n$, while the required transformation $L(I_n, I_n)$ involves exchanging the upper and lower n address elements in the address vector. The nature of the I/O transformation matrices prevents any effective transformation from being done in the read and write phases. The I/O phases 'gather' data to be permuted and 'scatter' the result of the permutation. In our algorithm, the I/O block size could be smaller than N, say $B = 2^b$, in which case the exchange $(b \ldots n-1) \leftrightarrow (n+b \ldots 2*n-1)$ can be done in the read and/or write phases. This reduces the number of address vector elements to be transformed in the in-memory permutation phase and might result in a reduction in the number of passes.

Our algorithm is formulated as shown below. The unit of each read and write is at least 2^r and 2^w elements respectively. Except in the first pass, the algorithm reads M elements in each read operation. In the first pass, the read and write phases transform the address vector elements $(w : n-1)$ to their appropriate positions. The remaining address vector elements are transformed in the in-memory permutation phase of all the passes and the I/O phases of the remaining passes.

Pre-conditions:

$$n \geq w$$

$$m \geq r \geq w$$

$$m > w$$

Parameters:

$$\begin{aligned}
s_0 &= \begin{cases} \min(m-r, w) & \text{if } r < n \\ \min(m-n, w) & \text{if } r \geq n \end{cases} \\
t &= \begin{cases} 1 & \text{if } s_0 = w \\ 1 + \lceil \frac{w-s_0}{m-w} \rceil & \text{otherwise} \end{cases} \\
s_i &= \min(m-w, w-s_0-(i-1)*(m-w)) \text{ if } 1 \leq i < t \\
z_i &= \begin{cases} 0 & \text{if } i = t-1 \\ m-(w+s_{i+1}) & \text{otherwise} \end{cases}
\end{aligned}$$

First pass$(i=0)$:

Case $1(r \geq n)$:

$$\begin{aligned}
R_0 &= I_{2n} \\
P_0 &= I_{n-s_0} \oplus L(I_{s_0}, I_{n-w} \oplus L(I_{w-s_0}, I_{s_0})) \\
W_0 &= L(I_{n-s_0}, I_{n-w+s_0-z_0}) \oplus I_{w+z_0}
\end{aligned}$$

Case $2(r < n)$:

$$\begin{aligned}
R_0 &= I_{n-s_0} \oplus L(I_{s_0}, I_{n-r}) \oplus I_r \\
P_0 &= I_{2n-(r+s_0)} \oplus L(I_{s_0}, I_{r-w} \oplus L(I_{w-s_0}, I_{s_0})) \\
W_0 &= L(I_{n-s_0}, I_{n-w+s_0-z_0}) \oplus I_{w+z_0}
\end{aligned}$$

Remaining passes:

$$\begin{aligned}
sp_i &= \sum_{j=0}^{i-1} s_j \\
R_i &= I_{2n} \\
P_i &= I_{2n-(w+s_i+z_{i-1})} \oplus L(I_{s_i}, I_{z_{i-1}} \oplus L(I_{w-sp_i-s_i}, I_{s_i})) \oplus I_{sp_i} \\
W_i &= I_{n-w} \oplus L(I_{s_{i-1}-z_{i-1}} \oplus L(I_{n-s_{i-1}-s_i}, I_{z_{i-1}}), I_{s_i-z_i}) \oplus I_{w+z_i}
\end{aligned}$$

With increasing memory size, modifying the I/O parameters provides diminishing improvements, unless it results in a reduction in the number of passes. Greater improvements can be obtained if the additional memory available is used to improve permutation time. Kaushik does an in-place in-memory transposition. Suh uses collect buffers to collect data to be written in each write operation. The locality of the permutation operation can be

Parameters: $n = r = w = t = 2$; $m = 3$; $s = \{1,1\}$; $z = \{0,0\}$

$$\begin{matrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{matrix}$$

Pass 0 (Case 1) :

$$R_0 = I_4 \qquad P_0 = I_1 \oplus L(I_1, L(I_1, I_1)) \qquad W_0 = L(I_1, I_1) \oplus I_2$$

$$\overset{R_0}{\Rightarrow} \begin{matrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{matrix} \quad \overset{P_0}{\Rightarrow} \begin{matrix} 0 & 4 & 2 & 6 \\ 1 & 5 & 3 & 7 \\ 8 & 12 & 10 & 14 \\ 9 & 13 & 11 & 15 \end{matrix} \quad \overset{W_0}{\Rightarrow} \begin{matrix} 0 & 4 & 2 & 6 \\ 8 & 12 & 10 & 14 \\ 1 & 5 & 3 & 7 \\ 9 & 13 & 11 & 15 \end{matrix}$$

Pass 1 ($sp_1 = 1$) :

$$R_1 = I_4 \qquad P_1 = I_1 \oplus L(I_1, I_1) \oplus I_1 \qquad W_1 = L(I_1, I_1) \oplus I_2$$

$$\overset{R_1}{\Rightarrow} \begin{matrix} 0 & 4 & 2 & 6 \\ 8 & 12 & 10 & 14 \\ 1 & 5 & 3 & 7 \\ 9 & 13 & 11 & 15 \end{matrix} \quad \overset{P_1}{\Rightarrow} \begin{matrix} 0 & 4 & 8 & 12 \\ 2 & 6 & 10 & 14 \\ 1 & 5 & 9 & 13 \\ 3 & 7 & 11 & 15 \end{matrix} \quad \overset{W_1}{\Rightarrow} \begin{matrix} 0 & 4 & 8 & 12 \\ 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \end{matrix}$$

Table 3.2: Illustration of our matrix transposition algorithm

improved by optimizations such a blocking. We use collect operations to perform the permutation, as this was empirically found to take less time than in-place permutation. Unlike Kaushik's and Suh and Prasanna's algorithms, the in-memory permutation in our algorithm moves larger blocks of data in each successive pass. This further reduces the in-memory permutation cost.

The transposition of a 4×4 array by our algorithm is illustrated in Table 3.2. The parameters are on the left hand side. The actual data layout after each transformation is shown on the right hand side.

3.3.6 Parallel Out-of-Core Matrix Transposition

In this section, the problem of transposing an out-of-core array distributed among multiple processors is discussed. Each processor has a local disk and the array is distributed among the processors in a row-blocked fashion. The required distribution of the transposed array among the processors is specified.

In the following discussion, we first formulate the representation of an array distributed among multiple processors. Then an algorithm is provided for redistributing out-of-core arrays in a parallel system. The array redistribution mechanism and the sequential transposition algorithm are combined to develop an out-of-core transposition algorithm for arrays distributed among multiple processors.

Formulation for Arrays Distributed among Multiple Processors

The arrays are assumed to be distributed in a regular fashion so that some of the elements in the address vector represent the processor identifier. This corresponds to a mapping of the elements of the array to a sequence of processors. A row-blocked distribution is obtained when the most significant elements in the address vector represent the processor identifier. A cyclic distribution is obtained when the least significant elements of the address vector represent the processor identifier.

We define the linear address vector of an element in the array to be the concatenation of the local address vector of the element (in the local disk) to the processor identifier. This view preserves the notion of contiguity of elements which differ in the least significant elements of the address vector, analogous to the sequential formulation. Hence the formulation can represent read and write thresholds in the address vector and access pattern that can take advantage of prefetching as well.

Given that the most significant elements in the linear address vector correspond to the processor identifier, the distribution of the array among multiple processors corresponds to choosing a set of elements in the address vector to become the most significant elements. Hence array distribution among multiple processors can be viewed as a permutation of the linear address space of the array. The identity for array distribution is I_{2n}, which corresponds to a row-blocked distribution. Any other distribution of data among processors is viewed as a permutation on the row-blocked distribution. For example, a cyclic distribution of an array among two processors corresponds to the following permutation:

$$\begin{pmatrix} 0 & 1 \\ I_{2*n-1} & 0 \end{pmatrix}$$

Array Redistribution Problem

The array redistribution problem is stated as follows: Given an array distributed among processors, represented by a permutation matrix, achieve a target distribution corresponding to a new permutation.

The array redistribution problem brings with it another cost factor in the form of communication. Communication cost varies linearly and is modeled as $T_s + l * T_b$, where T_s is the startup cost, l the message size and T_b the per-byte transfer cost. Depending on the parameters T_s and T_b, beyond a message size l, the transfer cost dominates the startup cost and the average per-byte cost converges to a constant. The message size beyond which there is little change in the communication cost is called the communication threshold 2^c. Note that as in the case of the read and write thresholds, the message size for a specific instance of an algorithm may be chosen below the threshold, if it cannot be improved upon. The communication characteristics of various systems have been widely studied and we do not discuss them here. For the following discussion, it is assumed that there are 2^p processors.

The uppermost p rows of any permutation matrix correspond to the elements that constitute the processor identifier. The least significant c elements of the address vector correspond to the communication threshold. The terms read, write, and communication thresholds will be used interchangeably to refer to the size of I/O and r, w ,and c least significant elements in the address vector, respectively. The reference will be clear from the context.

The formulation of the parallel redistribution problem involves four permutation matrices — read, write, in-memory permutation, and communication. Extending the template for the formulation of read, write, and in-memory permutation discussed in Section 3.3.3 to the parallel domain, we get

$$\begin{aligned} R_i &= I_p \oplus A_{2*n-r-p} \oplus I_r \quad r \leq m \\ P_i &= I_{2*n-m} \oplus B_m \\ W_i &= I_p \oplus C_{2*n-w-p} \oplus I_w \quad w \leq m \end{aligned}$$

which indicates that R_i, W_i and P_i cannot permute the elements corresponding to the processor identifier. Only communication can permute the elements corresponding to the processor identifier. The permutation corresponding to communication is of the form

$$C_i = D_{2*n-c} \oplus I_c$$

where D describes the permutations done by communication.

Note that there are some restrictions on C_i, similar to those on R_i, W_i, and P_i, as discussed in Section 3.3.3. C_i cannot permute between address elements corresponding to in-memory and out-of-memory data (the elements corresponding to the processor identifier are special and will be discussed below). Any permutation except those involving the processor identifier can be performed by P_i and W_i. Therefore, we place additional restrictions on C_i, so that it can only involve permutations required to change the processor identifier. In most practical systems, c is smaller than r and w; and we assume the same.

Array redistribution may involve permutations of three kinds. First, it may involve the exchange of address vector elements that are part of the processor identifier. This is achieved by an exchange of data between processors. Although an equivalent effect could be achieved by relabeling the processors, this cannot generally help us avoid inter-processor communication in practice, because multiple arrays are generally present and the processor relabeling will often force communication for other arrays.

The second kind of exchange occurs when elements within the communication threshold are to become part of the processor identifier. Any permutation involving the elements beyond the communication threshold is performed by an all-to-all personalized collective communication operation. If we have more than $m-c$ address elements within the communication threshold, that are to become elements corresponding to the processor identifier, then a sequence of in-memory permutation and communication operations are carried out. Each in-memory permutation operation moves as many elements from within the communication threshold to be beyond the threshold as possible. These elements are then made part of the processor identifier by a scatter operation. This process is repeated until there are no more elements in the least significant c address elements that are to be part of the processor identifier. Thus any element already part of the processor identifier or within the least significant m elements (memory size), that are to become part of the processor identifier, can be made part of the processor identifier in a single pass.

A more complicated operation is required when we need to permute the elements corresponding to the processor identifier with those beyond the least significant m elements.

This involves a collect operation by each processor. The difference in handling this case and the previous two cases is that in the previous two cases all processors perform the same operations throughout each pass. In this case, each processor collects all the data in memory from certain other processors in turn, in different iterations of the loop. Since all the collected data cannot be stored in memory, the data received from each processor must be written to disk. This interleaves communication and write operations, breaking the clear demarcation between the phases. Since this case essentially involves writing the data to disk, it is handled after the other two cases.

The above approach may not be the most efficient way of performing the array redistribution. In handling the last case, each processor might receive data from a different set of processors in different iterations. Each receive is separated by a write to disk. Hence the communication and write times cannot overlap, leading to very poor execution time especially when the number of processors is large. A more optimal implementation would be to schedule the communication among processors so that they overlap. A simple schedule would be for each processor to operate on data that has to be sent to one processor and then begin processing data to be sent to another processor. Each processor would be sending to and receiving data from a different processor, say in a ring topology, enabling overlap of communication and writing of data to disk. But this would modify the read and write access patterns by reordering of the reads and writes. The performance is not significantly impacted as the block size of I/O can been chosen to be large enough.

Hence all communication required to handle array redistribution can be done in a single pass. The implementation of this phase might involve a series of communications as just described. Henceforth we shall refer to C_i as the permutation effected on the linear address by the communication step and not delve into the implementation details.

Combining Array Redistribution and Sequential Matrix Transposition

In this section, we combine the mechanisms considered until now, to derive an algorithm for transposing out-of-core matrices which are distributed in a row-blocked fashion among multiple processors. Row-blocked distribution of data involves a permutation that is similar to transposition. Other regular data distributions can be characterized using other permutations. The approach presented applies to arbitrary regular distributions, but we only elaborate on the row-blocked case to illustrate the procedure involved.

The parallel version of the algorithm differs from the sequential version only in the first pass. Since array redistribution can be performed in a single pass, it is performed in combination with the first pass of the sequential algorithm. Subsequent passes are identical to running the sequential algorithm on all the processors. The first pass for the parallel algorithm is as follows:

- Read as in sequential case (R_i)
- Perform in-memory permutation as in sequential case P_i
- Perform array distribution, handling the different cases discussed above
- Perform any permutation need to regroup the data
- Write data to disk

The subsequent passes are identical to those in the sequential version. Thus the parallel version does not lead to an increase in the number of passes in the form of additional reads or writes. The formulation for the first pass is as shown below:

Pre-conditions:

$$\begin{aligned} n &\geq w \\ m &> r \geq w \\ m &> w \end{aligned}$$

Parameters

$$\begin{aligned}
T &= \prod_{t-1}^{0} T_i \\
T_i &= \begin{cases} W_i P_i R_i & \text{if } i > 1 \\ W_i P_i' H_i P_i R_i & \text{otherwise} \end{cases} \\
s_0 &= \begin{cases} \min(m-r-1, w) & \text{if } r < n \\ \min(m-n-1, w) & \text{if } r \geq n \end{cases} \\
s' &= \begin{cases} p-(n-w) & \text{if } p > (n-w) \\ 0 & \text{otherwise} \end{cases} \\
t &= \begin{cases} 1 & \text{if } s_0 + s' = w \\ 1 + \lceil \frac{w-s_0-s'}{m-w} \rceil & \text{otherwise} \end{cases} \\
k &= (w - s_0 - s') \bmod (m-w) \\
s_i &= \begin{cases} k & \text{if } i = t-1 \text{ and } k > 0 \\ m-w & \text{otherwise} \end{cases}
\end{aligned}$$

Case 1: $(r \geq n)$

$$\begin{aligned}
R_0 &= I_{2n} \\
P_0 &= I_{2n} \\
H_0 &= L(I_p, L(I_{n-p}, I_p)) \oplus I_{n-p} \\
P_0' &= I_{n-s_0} \oplus L(I_{s_0}, I_{n-w} \oplus L(I_{w-s_0}, I_{s_0})) \\
W_0 &= I_p \oplus L(I_{n-p-s_0}, L(I_p, I_{n+s_0-p-w})) \oplus I_w
\end{aligned}$$

Case 2: $(r < n \wedge p \leq (n-w))$

$$\begin{aligned}
R_0 &= I_{n-s_0} \oplus L(I_{s_0}, I_{n-r}) \oplus I_r \\
P_0 &= I_{2n-(r+s_0)} \oplus L(I_{s_0}, I_{r-w} \oplus L(I_{w-s_0}, I_{s_0})) \\
H_0 &= L(I_p, L(I_{n-p-s_0}, I_p)) \oplus I_{n+s_0-p} \\
P_0' &= I_{2n} \\
W_0 &= I_p \oplus L(I_{n-p-s_0}, L(I_p, I_{n+s_0-p-w})) \oplus I_w
\end{aligned}$$

Parameters: $n = m = t = 2; r = w = p = 1; s = \{0,1\}; s' = 0$

$$\begin{array}{cccc} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ \hline 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{array}$$

Pass 0 (Case 2):

$$\begin{array}{c} R_0 = I_4 \\ P_0 = I_4 \end{array} \qquad \begin{array}{c} H_0 = L(I_1, L(I_1, I_1)) \\ \oplus I_1 \end{array} \qquad \begin{array}{c} P_0' = I_4 \\ W_0 = I_1 \oplus L(I_1, I_1) \oplus I_1 \end{array}$$

$$\xRightarrow{R_0P_0} \begin{array}{cccc} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ \hline 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{array} \xRightarrow{H_0} \begin{array}{cccc} 0 & 1 & 8 & 9 \\ 4 & 5 & 12 & 13 \\ \hline 2 & 3 & 10 & 11 \\ 6 & 7 & 14 & 15 \end{array} \xRightarrow{P_0'W_0} \begin{array}{cccc} 0 & 1 & 4 & 5 \\ 8 & 9 & 12 & 13 \\ \hline 2 & 3 & 6 & 7 \\ 10 & 11 & 14 & 15 \end{array}$$

Pass 1:

$$\begin{array}{c} R_1 = I_4 \end{array} \qquad \begin{array}{c} P_1 = I_2 \oplus L(I_1, I_1) \end{array} \qquad \begin{array}{c} W_1 = I_1 \oplus L(I_1, I_1) \oplus I_1 \end{array}$$

$$\xRightarrow{R_1} \begin{array}{cccc} 0 & 1 & 4 & 5 \\ 8 & 9 & 12 & 13 \\ \hline 2 & 3 & 6 & 7 \\ 10 & 11 & 14 & 15 \end{array} \xRightarrow{P_1} \begin{array}{cccc} 0 & 4 & 1 & 5 \\ 8 & 12 & 9 & 13 \\ \hline 2 & 6 & 3 & 7 \\ 10 & 14 & 11 & 15 \end{array} \xRightarrow{W_1} \begin{array}{cccc} 0 & 4 & 8 & 12 \\ 1 & 5 & 9 & 13 \\ \hline 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \end{array}$$

Table 3.3: Illustration of our parallel matrix transposition algorithm

$$\begin{aligned}
\text{Case 3:} \quad & (r < n \wedge p > (n - w)) \\
R_0 &= I_{n-s_0} \oplus L(I_{s_0}, I_{n-r}) \oplus I_r \\
P_0 &= I_{2n-(r+s_0)} \oplus L(I_{s_0}, I_{r+p-n} \oplus L(I_{n-p-s_0}, I_{s_0})) \\
H_0 &= L(I_n - w \oplus L(I_{p-(n-w)}, I_{n-p-s_0}), I_p) \oplus I_{n-p+s_0} \\
P_0' &= I_{2n-w-s_0} \oplus L(I_{p-(n-w)}, I_{s_0}) \oplus I_{n-p} \\
W_0 &= I_p \oplus L(I_{2n-p-w-s_0}, I_{s_0}) \oplus I_w
\end{aligned}$$

There are some noticeable differences in the first pass as compared to that in the sequential algorithm. H_0 represents the array redistribution phase. The first pass consists of five phases. There are two in-memory permutation steps, P_0 and P'_0, that prepare data for communication and regroup the data before writing to disk. This could involve a series of interleaved permutation and communication steps, where the communication steps satisfy the communication threshold. Communication requires buffers to store the received data, in addition to the data read from disk, which might be sent to another processor in parallel. Thus the amount of memory available should be at least twice the read block size chosen. An increase in the number of processors implies an increase in the total available memory. If the number of processors is large enough, the communication phase can contribute to permuting the address elements within the write threshold. This factor is represented by s'. When the number of processors is large enough to contribute to permutation of the linear address, the communication and in-memory permutations involved are different from when it is not. The formulation handles all the different cases.

The transposition of a 4×4 array is illustrated in Table 3.3. The array is distributed in a row-blocked fashion among 2 processors. The transposed array is also required to be in a row-blocked distribution. In terms of data, the top half of the matrix is stored in the first processor's disk, the bottom half on second processor's disk. The parameters of the algorithm are shown on the left hand side of the table. The actual data layout is shown on the right hand side. The algorithm requires two passes to transpose the array. In the first pass, no elements within the write block size are permuted and no in-memory permutation is done. In the illustration, these permutations are combined with other phases to simplify the figures, as they are just identity transformations. Upon completion of the first pass, the elements have been redistributed to the target processors. In the second pass, each processor permutes the array independently to arrive at the transposed form. Note that the reads and writes conform to the read and write block sizes, thus ensuring high bandwidth for disk I/O.

3.3.7 Experimental Evaluation

We now discuss experimental results obtained from evaluating the parallel transposition algorithm on the amd-osc and ia64-osc clusters. Both clusters use the Myrinet [13] interconnection network. The implementation was out-of-place and used an auxiliary array.

The transposition times for different memory sizes and numbers of processors were measured. Tables 3.4 and 3.5 show the transposition times on ia64-osc for array sizes of 16GB (N=64K) and 64GB (N=128K). Tables 3.6 and 3.7 show the transposition times on amd-osc for the same array sizes.

In both systems the read threshold was much higher than N. So the execution time was determined primarily by the write threshold. Increasing the memory available decreases the number of I/O operations. If I/O operations were an effective measure of performance, doubling the memory size should halve the execution time. But the execution time improves little with increase in memory size, except when the larger memory size leads to a reduction in the number of passes. Reduction in the number of passes is accompanied by a significant reduction in the total execution time. This can be seen, for example, in the transition from 32MB to 64MB on one processor in Table 3.6. The slight improvement seen with the increase in memory size is due to a reduction in the stride of writes. The write block size is reduced to be below the write threshold if it can reduce the number of passes and hence the total execution time. This is the case for 64MB memory on the one processor in Table 3.6. In certain cases, the stride of write is so large as to wrap around

#procs	Memory size (MB)					
	16	32	64	128	256	512
1	3406	3322	2265	2230	2003	2079
2	1536	1127	962	949	984	1006
4	740	542	484	483	475	474

Table 3.4: Parallel matrix transposition time, in seconds, on ia64-osc. Array size is 16GB (N=64K)

and result in the writing of adjacent blocks before earlier written blocks have been flushed to disk. This leads to larger write block sizes and hence shorter total execution time. This trend can be especially seen in Table 3.6 at the transition in the number of passes, when the write block size is reduced to avoid an increase in the number of passes.

The parallel algorithm scales well with an increase in the number of processors. A slightly super-linear speedup can be seen in some cases. This is due to improved locality in I/O. Note that for an array size of 16GB and for four processors, the portion of each array in a processor is 4GB, equal to the memory size in the Itanium 2 cluster. But since there are three arrays the arrays are not fully cached in memory, making the results dependent on the operating system caching mechanism. In some cases, an increase in the number of processors reduces the number of passes, thus significantly reducing the execution time. This effect can be observed in Table 3.6 for a memory size of 32MB, when the number of processors is increased from one to two.

#procs	Memory size (MB)					
	16	32	64	128	256	512
4	3448	3252	3213	2102	2907	2801
8	1470	1533	1469	921	985	1007

Table 3.5: Parallel matrix transposition time, in seconds, on ia64-osc. Array size is 64GB (N=128K)

#procs	Memory size (MB)					
	16	32	64	128	256	512
1	7443	7386	3344	4254	4374	4223
2	3865	2098	2179	2253	2333	2207
4	1971	981	1142	1131	1165	1122
8	995	583	549	688	638	560

Table 3.6: Parallel matrix transposition time, in seconds, on amd-osc. Array size is 16GB (N=64K)

#procs	Memory size (MB)					
	16	32	64	128	256	512
4	8122	6365	4948	3959	3855	3923
8	3523	3469	2695	2167	2046	1855

Table 3.7: Parallel matrix transposition time, in seconds, on amd-osc. Array size is 64GB (N=128K)

3.4 Out-of-core Matrix Reblocking

3.4.1 Background

This work is done in the context of the Global Arrays programming suite. The Global Arrays suite [85] provides a set of inter-operable programming models, each at a different level of abstraction. At the lowest level is MPI, a distributed-memory programming model with message passing for two-sided communication. Though MPI is not part of the suite, it is fully inter-operable with the abstractions provided in the suite, and is an integral part of the hierarchy of abstractions presented to the user.

The Aggregate Remote Memory Copy Interface (ARMCI) library [83] provides a distributed-memory view with one-sided access to remote data. It has a rich set of primitives for non-blocking operations, and contiguous and non-contiguous data transfers optimized to hide latency. ARMCI forms the underlying communication layer for a number of compile/runtime systems, including Co-Array Fortran [24], GPSHMEM [90], and Global Arrays.

The next higher level is the Global Arrays (GA) library. [84, 86]. GA exposes a global view of a dense multi-dimensional array distributed amongst the local memories of processors. It provides a shared-memory programming model in which data locality is explicitly managed by the programmer. Explicit function calls are used to transfer data between global address space and local storage. It is similar to distributed shared-memory models in providing an explicit acquire-release protocol, but differs with respect to the level of explicit control in moving blocks of data in multidimensional arrays between remote global storage and local storage. The functionality provided by GA has proved useful in the development of large scale parallel quantum chemistry suites such as NWChem [47] (which contains over a million lines of code), adaptive mesh refinement codes such as NWPhys/NWGrid (www.emsl.pnl.gov/nwphys) and applications in other areas [85].

The Disk Resident Arrays (DRA) model [86] extends the GA programming model to secondary storage. It provides a disk-based representation for multi-dimensional arrays and operations to transfer blocks of data between global arrays and disk resident arrays.

Global Arrays allows the user to assume a shared-memory programming model, while simultaneously supporting mechanisms to query for and manipulate local data. Thus it also enables incremental optimizations. ARMCI, GA, and DRA provide a unified programming model for handling different levels of the memory hierarchy in which the user controls the location of data in the memory hierarchy. This has been shown to achieve high performance, while being a simpler programming model than message passing.

3.4.2 Problem Definition

Internally, the data in a DRA is stored in a blocked fashion. When a DRA is created, a typical request shape/size can be specified. This is used to determine the shape of the basic layout block or “brick”. The shape of the brick is chosen to match the specified access shape. The size of the brick is chosen as a compromise between two competing objectives: 1) optimize disk I/O bandwidth – this requires that the brick size be large enough to amortize the disk seek time and 2) minimize wastage of disk I/O – since I/O is done in units of the basic block (brick), small bricks imply less wastage at the boundaries of the DRA regions being read/written.

An application might have an access pattern that is very different from the organization of the DRA on disk. This can happen when an application uses the output of another program, or because different phases of the same program use different access patterns.

This can be handled by creating another copy of the disk resident array to match the new request size and transformed dimensions.

We have implemented the copy routine, referred to as *NDRA_Copy*, together with dimension permutation. The routine takes as input the source and target DRA handles and the dimension permutation to be performed. Henceforth, the data in the DRA corresponding to the dimensions of blocking in the source and target arrays are referred to as the source and target blocks, respectively.

The disk array layout transformation problem we consider here is a generalization of the out-of-core matrix transposition problem. Most existing solutions to the problem, including our solution presented earlier, assume the array dimensions and the memory size to be powers-of-2. This assumption, coupled with the fact that the required transformation is a transposition, allows different steps in the re-blocking process to operate on disjoint sets of data. In each step, the set of data read into memory form an integral number of write blocks, which are written out. So no data is retained across steps during the transposition. When arbitrary blocking, array dimensions and memory sizes are to be handled, it may not be possible to process and write out all the data read into memory in a given step. Some data either needs to be discarded and re-read, increasing the I/O cost, or needs to be retained, increasing the memory requirement. The memory cost for retaining the data unused from a step depends on the order of traversal of dimensions, and hence is not straight forward. The out-of-core transposition algorithms involve I/O of blocks of data at specific strides, which is fixed for a pass. This regularity allows better prediction of the I/O cost. The in-memory permutation of data can be modeled as a bit-permutation on the linear address space of the data stored in disk. This provides a regular structure to the in-memory computation.

3.4.3 Algorithm Design

The disk array layout transformation problem is modeled as an I/O optimization problem. The total I/O cost is to be minimized, subject to the amount of physical memory available. In the ensuing discussion, we shall consider an n-dimensional matrix of dimensions $\langle d_1, \ldots, d_n \rangle$, with blocks of shape $\langle s_1, \ldots, s_n \rangle$. The target matrix has the same ordering of dimensions as the source but is blocked using blocks of shape $\langle t_1, \ldots, t_n \rangle$. The source and target bricks are assumed to be of size that is large enough for efficient access from/to disk. DRA typically uses a brick size of around 1 Mbyte. Reads from the source disk array are assumed to be in units of the source brick, and writes to the target disk array are done in units of the target brick.

Solution Approach

If feasible, a single-pass solution (in which each element is read and written exactly once) would provide the minimum I/O cost. But the memory requirement for a single-pass solution might exceed the physical memory available. In this case, we either need to choose a multi-pass solution or perform redundant I/O in one pass. In this sub-section, we present the intuition behind the design of our algorithm. We begin with a basic single-pass algorithm and determine its I/O and memory cost. We then incrementally improve the single-pass algorithm to lower the memory requirement and/or the I/O cost. The multi-pass solution is discussed in a subsequent sub-section.

Consider the region $\langle 0 - LCM(s_1, t_1), \ldots, 0 - LCM(s_n, t_n) \rangle$. This region contains an integral number of source and target blocks along all the dimensions. Thus the data in the source matrix from this region maps onto complete blocks in the target matrix. This region can be processed independent of other such regions, without any redundant I/O. We shall refer to such regions as *LCM blocks*. If the amount of physical memory were large enough to hold an LCM block, then a single-pass solution is clearly possible – read in source blocks contained in an LCM block into memory, construct the target blocks corresponding to the data in memory, and write them to the target array. The I/O cost is defined as the I/O required per element of the source array. This algorithm has the minimum I/O cost of one read and one write per element of the source array. Assuming the read and write operations are equivalent the I/O cost is two units per element.

The memory cost is the size of the LCM block. Since arbitrary re-blocking needs to be supported, the source and target block sizes could have arbitrary dimensions (provided their total size corresponds to a reasonable block size for I/O on the target file system). Hence the LCM block can be arbitrarily large and might not fit in physical memory. We can improve the single-pass algorithm to handle this scenario without increasing the I/O cost. Instead of reading entire LCM blocks into memory, the algorithm reads in a set of blocks of data from the source matrix and writes out those target blocks that can be completely constructed from the data available in memory. Any data in memory that cannot be used to construct a complete target block is retained in memory. Any source block in an LCM block contributes to target blocks within the same LCM block. Hence no data needs to be retained across LCM blocks. The algorithm processes all the data in one LCM block before processing any other LCM block. The algorithm requires enough memory to retain unused data and read in additional data for processing. The additional data read into memory for processing must be enough to write at least one target block to disk. This is referred to as the Max block and corresponds to $\langle M_1, \ldots, M_n \rangle$ where $Max_i = \lceil \max(s_i, t_i)/s_i \rceil * s_i$. The algorithm traverses each LCM block along each of the dimensions and processes data in units of the Max block. The buffer to store the unused data is partitioned into one buffer per dimension. Unused data from a Max block along a dimension needs to be retained until the adjacent Max block along that dimension is processed. Thus the amount of unused data to be retained depends on the order of traversal of dimensions. Along the dimension traversed first, only data unused from the last processed Max block needs to be stored. Other dimensions require more data to be retained. A static memory cost model is used, in which the sizes of buffers used to store data is determined before the transformation begins. The maximum memory required to perform the transformation is the sum of the size of the Max block and the sizes of the buffers.

$$\text{MemCost} = \sum_{i=1}^{n} \text{bsize}_i + \prod_{i=1}^{n} Max_i$$

where bsize_i represents the size of buffer to store unused data along the i-th dimension.

Let $\langle T_1, \ldots, T_n \rangle$ be the order of traversal of dimensions. The unused data along a dimension (say T_i) is an n-dimensional region. For a given dimension i, the size of this region along dimension j can be as much as $\text{LCM}(s_{T_j}, t_{T_j})$ for $j < i$, but is bounded above by Max_{T_j} for $j > i$. Hence, the size of the buffer to store the unused data along a dimension T_i is bounded by

$$\text{bsize}_{T_i} = \prod_{j=1}^{n} S_j$$
$$S_j = \begin{cases} \text{LCM}(s_{T_j}, t_{T_j}) & \text{if } j < i \\ U_{T_j} & \text{if } j = i \\ Max_{T_j} & \text{if } j > i \end{cases}$$

where U_i be the maximum unused data that needs to be stored along dimension i. Since U_i must be smaller than both s_i and t_i, and for every s_i elements along dimension i brought into memory, at least $\text{gcd}(s_i, t_i)$ elements must be written out, we have

$$U_i = \min(s_i, t_i) - \text{gcd}(s_i, t_i)$$

For a two-dimensional array, the memory cost due to the unused buffers is $U_1 * Max_2 + \text{LCM}(s_1, t_1) * U_2$ if dimension 1 is traversed first; otherwise, it is $U_2 * Max_1 + \text{LCM}(s_2, t_2) *$

```
function MemCost($s$,$t$,$templ$)
    Input: Source and target block sizes, and template size
    Output: Total memory cost, dimension traversal order
    for each dimension $i$ do
        $L_i = \mathrm{lcm}(s_i, t_i)$
        $U_i = \min(s_i, t_i) - \gcd(s_i, t_i)$
        $M_i = \lceil(\max(s_i, t_i)/s_i)\rceil * s_i$
    Sort dimensions into array T such that $(\forall i, j)\ i < j \Rightarrow (U_{T_i} * M_{T_j} + L_{T_i} * U_{T_j} < U_{T_j} * M_{T_i} + L_{T_j} * U_{T_i})$
    memCost=0
    for each dimension $i$ do
        pdt= $U_{T_i}$
        for each $j < i$ do
            pdt = pdt $* L_{T_j}$
        for each $j > i$ do
            pdt = pdt $* M_{T_j}$
        memCost $+=$ pdt
    return $\langle$memCost,$T\rangle$
```

Algorithm 3.6: Algorithm to determine the memory cost for a given template size

U_1. In an n-dimensional array, the traversal order is determined by sorting the dimensions by comparing these expressions.

As can be seen from the above formulae, the sizes of the unused buffers is proportional to the LCM block dimensions. This could lead to situations in which the memory requirement still exceeds the available memory. In this case, there are two options to be considered. A multi-pass solution could be determined, which is discussed later, or a single-pass solution that performs redundant read of data can be designed.

We propose a single-pass algorithm that differs from the discussion above in one respect. Instead of traversing an entire LCM block, a smaller template is chosen. No unused data is stored across templates. A template is an integral number of write blocks along all dimensions. There is no redundant read within a template. But unlike LCM blocks, templates might have source blocks on their boundaries that straddle across two templates.

This results in redundant reads across templates, increasing the I/O cost. The memory cost is reduced and is given by:

$$\text{MemCost} = \sum_{i=1}^{n} \text{bsize}_i + \prod_{i=1}^{n} Max_i$$

$$\text{bsize}_{T_i} = \prod_{j=1}^{n} S_j$$

$$S_j = \begin{cases} \text{templ}_{T_j} & \text{if } j < i \\ U_{T_j} & \text{if } j = i \\ Max_{T_j} & \text{if } j > i \end{cases}$$

where templ_i represents the size of the template along the i-th dimension.

The minimum template size corresponds to a target block. In this case, the memory requirement is reduced to a Max block. Thus the necessary condition for the existence of a single-pass solution is that the Max block fit in memory.

The I/O cost is multiplicative along the dimensions. Within an LCM block, the number of source blocks that need to be reread is the number of templates minus one, which is $(\lceil \text{LCM}(s_i, t_i)/\text{templ}_i \rceil - 1)$. Therefore, the I/O cost of re-blocking is given by templ_i is

$$\text{IOCost} = \prod_{i=1}^{n} \text{IOCost}_i$$

$$\text{lcm}_i = \text{LCM}(s_i, t_i)$$

$$\text{IOCost}_i = \frac{s_i * \left(\left\lceil \frac{\text{lcm}_i}{\text{templ}_i} \right\rceil - 1 \right) + \text{lcm}_i}{\text{lcm}_i}$$

In reality, the LCM along a dimension might be larger than the length of the array along the dimension, in which case we replace the LCM by the array dimension. Note that the array dimensions are not considered while determining U_i. Hence, U_i does not provide an exact estimate, but only an upper bound on the memory requirement. Even though this approach might increase the I/O cost for a single pass, the total I/O cost could be reduced due to a reduction in the number of passes.

```
function SinglePassSolution($s$,$t$,MemoryLimit)
    Input: Source and target block sizes, and memory Limit
    Output: I/O cost, template size, and dimension traversal order
    $(\forall i)$ templ$_i$ = lcm($s_i, t_i$)
    $\langle$cost,T$\rangle$ = MemCost(s,t,templ)
    while cost $>$ MemoryLimit do
        $(\forall i)$ templ$_i$ = templ$_i$ - 1
        if $(\exists i)$ templ$_i \leq 0$ then
            return $\langle \infty, \text{templ}, T \rangle$
        $\langle$cost,T$\rangle$ = MemCost(s,t,templ)
    Adjust the template size so that increasing the template size along any dimension makes it
        infeasible
    while true do
        Among adjacent template sizes choose the one that has the maximum rate of decrease
            in I/O cost to increase in memory cost
        Determine a feasible template, templ' that leads to the least increase in disk I/O cost
            from the chosen template
        if DiskCost(templ) $\leq$ DiskCost(templ') then
            return $\langle$DiskCost(templ), templ, $T\rangle$
        templ = templ'
        $\langle$cost,T$\rangle$ = MemCost(s,t,templ)
```

Algorithm 3.7: Algorithm to determine template size for a single-pass solution.

Template Determination for Single-Pass Solution

Both the I/O cost and the memory cost are affected by the choice of the template. The template is a set of write blocks along all the dimensions. It can range in size from one write block to an LCM block. For re-blocking an *n*-dimensional array, the template needs to be determined from an *n*-dimensional solution space. A template is a feasible solution if its processing does not require more memory than available. The algorithm exploits the characteristics of the solution space and the optimization function.

Consider a template *A*. An enclosing template is defined as a template that is at least as large as the given template in all the dimensions. Let *B* be an enclosing template of *A*. From the memory cost equations, it can be seen that the memory required to process

A cannot exceed that required to process *B*. Conversely, processing *B* requires at least as much memory as processing *A*. This implies that once a template has been determined to require more memory than available (an infeasible solution), no enclosing templates needs to be considered. This relation separates the solution space into a feasible and an infeasible solution space (where the surface of separation approximates to a hyperbola when $n = 2$).

The I/O cost has a similar characterization. The I/O cost equation shows that decreasing the template size along any dimension increases the I/O cost. Thus the I/O cost of template *A* is at least as much as that of template *B*. This implies that when searching through the solution space, no template that is enclosed by a feasible template needs to be considered. Thus the optimal solution resides on the surface separating the feasible and infeasible solution spaces.

Our algorithm to determine the template for a single-pass solution involves three phases. The algorithm begins with the LCM block as the template and tests for feasibility. If an LCM block is the feasible solution, it is chosen as the template. Otherwise, a solution is chosen that is just feasible, i.e., increasing the template size along any dimension violates the memory constraint. This is a solution on the boundary between the feasible and infeasible solution spaces and hence is a candidate solution. From this solution, we perform a steepest descent to arrive at a local minimum in the search space. Note that other optimization algorithms that can optimize on a surface can be used.

Multi-pass Solution Determination

When a single-pass solution does not exist or is too expensive, a multi-pass solution is chosen by determining intermediate block sizes. An intermediate disk-based array is used to store the intermediate results. Hence, additional disk space equal to the size of the

```
function MultiPassSolution($s$, $t$, arraySize, MemoryLimit)
    Input: Source and target block sizes, array size, memory limit
    Output: Total I/O cost and passes
    $\langle$sCost,sTemplate,sDimOrder$\rangle$ = singlePassSolution($s$,$t$, MemoryLimit)
    if sCost = 2 * arraySize then                         ▷ Minimum I/O Cost possible
        return $\langle$ sCost,list($\langle$sTemplate,sDimOrder$\rangle$)$\rangle$
    $(\forall i) c_i = \lfloor \sqrt{s_i * t_i} \rfloor$
    $(\forall i) c1_i = s_i^{2/3} * t_i^{1/3}$
    $(\forall i) c2_i = s_i^{1/3} * t_i^{2/3}$
    $\langle$cost1a,passes1a$\rangle$ = MultiPassSolution($s$, $c$,arraySize,MemoryLimit)
    $\langle$cost1b,passes1b$\rangle$ = MultiPassSolution($c$, $t$,arraySize,MemoryLimit)
    $\langle$cost2a,passes2a$\rangle$ = MultiPassSolution($s$, $c$1,arraySize,MemoryLimit)
    $\langle$cost2b,passes2b$\rangle$ = MultiPassSolution($c$1, $c$2,arraySize,MemoryLimit)
    $\langle$cost2c,passes2c$\rangle$ = MultiPassSolution($c$2, $t$,arraySize,MemoryLimit)
    cost = min (sCost,cost1a+cost1b,cost2a+cost2b+cost2c)
    if cost = sCost then
        passes = list($\langle$sTemplate,sDimOrder$\rangle$)
    else if cost = cost1a+cost1b then
        passes = concatLists(passes1a,passes1b)
    else if cost = cost2a+cost2b+cost2c then
        passes = concatLists(passes2a,passes2b,passes2c)
    return $\langle$cost,passes$\rangle$
```

Algorithm 3.8: Algorithm to determine a multi-pass solution

arrays is required. The multi-pass solution proceeds as repeated execution of the single-pass algorithm, for the source and target block sizes determined for that pass. The source block size of the first pass is the block size of the source array. The target block size of the last pass if the block size of the target array. The skew between the source and target block sizes decreases as the multi-pass solution proceeds from one pass to the next. The intermediate block size are chosen to effect the maximum re-blocking possible with the available memory.

A simple heuristic is used to determine the intermediate tile sizes for the multi-pass solution. Two candidate intermediate block sizes are considered. The first candidate intermediate block size is the geometric mean of the source and target block sizes. This block size is “equidistant” from the source and target block sizes. This can be an effective intermediate block size of for solutions with an even number of passes. The second intermediate block size is, in fact, a pair of block sizes. Let s_i and t_i be the source and target block sizes along dimension i. The intermediate block sizes chosen are $s_i^{2/3} * t_i^{1/3}$ and $s_i^{1/3} * t_i^{2/3}$. This pair of intermediate block sizes can be effective for solutions with an odd number of passes. These two options allow a more refined search for intermediate block sizes. Without the second choice, any solution that requires an odd number of passes, each transforming to an intermediate block “equidistant” from the previous one, might be harder to achieve. Higher order intermediates were not considered as solutions with a larger number of passes seldom occur in practice and can be handled by a combination of these choices.

Once the intermediate block(s) are determined, the multi-pass solution is determined recursively for transforming from source to intermediate, and intermediate to target block sizes. In the case of two intermediate blocks, the transformation between the intermediate blocks is determined as well. The algorithm for determining the multi-pass solution is shown in Algorithm 3.8.

Consider an instance of the matrix re-blocking problem in which the source and target arrays are blocked as $\langle 32, 9\rangle$ and $\langle 5, 16\rangle$, respectively. The array dimensions are much larger than the blocking and hence are not considered. The Max block is $\langle 32, 16\rangle$ and the unused data along each dimension is bounded by $\langle 4, 8\rangle$. The solution to the re-blocking problem depends on the memory available. An LCM block contains $LCM(s_1,t_1) * LCM(s_2,t_2)$

```
1:  **Input**: Source and target DRAs [$d_s$] and [$d_t$]
2:  **Output**: $d_t$ contains the data in $d_s$
3:  Determine the multi-pass solution
4:  Create a file as an intermediate. Use space in $d_t$ as the other intermediate
5:  **for each** pass **do**
6:      Determine source and target files for this pass (so that the target in the last pass is $d_t$)
7:      Allocate memory for unused buffers along each dimension, the buffer to contain the Max
          block, and a write block
8:      **for each** template $t$ **do**
9:          **while** Max blocks remain to be processed **do**
10:             **Read** the next Max block into memory from the source
11:             Construct complete write blocks from Max block and unused buffers
12:             **Write** the constructed complete write blocks to target
13:             If Max block contains unused data corresponding to current template, store it into
                  unused buffers
14: Delete the temporary file
```

Algorithm 3.9: Algorithm for sequential implementation of layout transformation

= 23040 elements. When enough memory is available to hold an LCM block, the re-blocking can be performed by reading in an entire LCM block and writing out the target blocks. But if the memory can hold $U_2 * Max_1 + \mathrm{LCM}(s_2, t_2) * U_1 + Max_1 * Max_2$=1344 elements, it is sufficient to hold all unused data when an LCM block is processed. The second dimension is traversed first in the re-blocking procedure. If the memory available is lesser, say enough to hold just 900 elements, a single-pass solution with a template size of $\langle 120, 6\rangle$ elements is used for the re-blocking. When the memory size is 800, a two-pass solution with an intermediate tile size of $\langle 12, 12\rangle$ is determined. The template for the first pass is $\langle 96, 12\rangle$, and that for the second pass is $\langle 60, 48\rangle$.

3.4.4 Implementation

A pseudo-code for the sequential implementation, using file I/O, is shown in Algorithm 3.9. The number of passes and the intermediate block sizes for each pass are first determined using the multi-pass solution algorithm in Algorithm 3.8. The target DRA and an additional temporary file are used to store the intermediate data. The input file in the first pass is the one corresponding to the source DRA. The input and output files for each pass are chosen in such a way that the output file in the last pass is the file corresponding to the target DRA. The computation proceeds in a sequence of passes. The buffers to hold the unused data and the Max block are initialized. In each pass, the templates are processed one after another. The data corresponding to each template is traversed in units of Max block, in the predetermined order. In each step, a Max block is read into memory, complete write blocks are constructed and written into the output file. Reading a Max block from disk involves a sequence of I/O operations one for each brick in the Max block. If the Max block contains any unused data corresponding to the current template, it is stored in the unused buffers. If the Max block is only partially present in the current template (i.e., some of it corresponds to write blocks in another template), the data not relevant to the current template is discarded. Construction of the complete write blocks involves determining the regions of the read blocks to be combined, locating the regions from the buffers, and patching the data onto a temporary buffer. The data in the temporary buffer is then written to disk.

Implementation Choices

We needed a parallel implementation that can handle the different forms of disk arrays, in particular arrays on local disks and on a shared file system. Various alternatives in obtaining a parallel implementation of the algorithm were considered. The alternatives differed in the the level of abstraction utilized and the granularity of parallelism exploited.

At the coarsest level of parallelism, each template can be processed independently and hence can be assigned to a different process. Each process handles the next available template, which is determined at runtime. This provides automatic load-balancing. Since the processes operate on disjoint sets of data, a low-level abstraction is required. GA/DRA requires a collective operation to perform I/O on the disk array, which is not suitable for template-level parallelism. The absence of one-sided access to the data on the remote disk necessitates co-ordination of the computation amongst the different processors. This requires a load-balancing scheme different from the process-next-template scheme.

Another significant drawback of utilizing template-level parallelism is that orchestration of the computation amongst all the processors can utilize the global memory for processing. This can potentially reduce the number of passes, by allowing a greater component of the transformation to be done in each pass. Thus, it is advantageous to have all the processes co-operate in transforming each template. Parallelism in the form of distributed ownership of the bricks by the I/O processes, those that perform I/O, is exploited. We redefine the Max block in each step to be such that enough complete write blocks can be constructed to utilize all the available I/O processors.

The co-ordination amongst the processes can be achieved either using MPI and file I/O or using GA/DRA. Using MPI and file I/O provides greater flexibility and predictability to the computation. This could allow tuning the implementation to the specific environment. Alternatively, GA/DRA abstracts away the complexity in dealing with file offsets, packing and unpacking of data, and message passing. That GA allows the use of message passing, in particular MPI method calls, on both GA and non-GA data in a GA/DRA program enables incremental tuning of the implementation. A GA/DRA implementation can be further tuned using MPI and file I/O if such tuning can improve performance. When the tuning does not improve performance, a more maintainable code is available. The lessons learnt from the tuning process can help in making further improvements to the GA/DRA model.

To illustrate the incremental tuning in the GA/DRA model, let us consider two possible optimizations. The GA/DRA implementation reads data from a disk array into a global array in memory. The data is processed in the global array and written back into the disk array. The data could be read into local memory, copied into a global array, and then writen from the global array to a disk array. This could reduce the communication overhead or schedule communication in an intelligent manner. Alternatively, data can be read from a disk array into a global array, and each processor can copy the data onto its local memory and write to the block it handles.

One attendant disadvantage of using GA/DRA for the operation is the increased disk space requirement. At any point in the computation, space is required for the source and target arrays of the current pass and the ultimate source and target arrays of the transformation. The input array is assumed to be read-only. The output array is unused until the last pass, and can potentially be utilized. But accessing the space allocated to the target array via the DRA induces a blocking that is usually incompatible with the blocking of the intermediate data. Operating at the file I/O level, one can bypass the blocked view and directly access the space. A simple extension to the GA/DRA framework, in which multiple disk arrays can use the same file (analogous to the union type in C) could be provided to allow different blocking views to the same data space on disk.

Another optimization is possible when operating at the file I/O level. Instead of operating on the entire array on a pass before proceeding to the next pass, each template can be processed through all the passes and written into the final array before processing the next

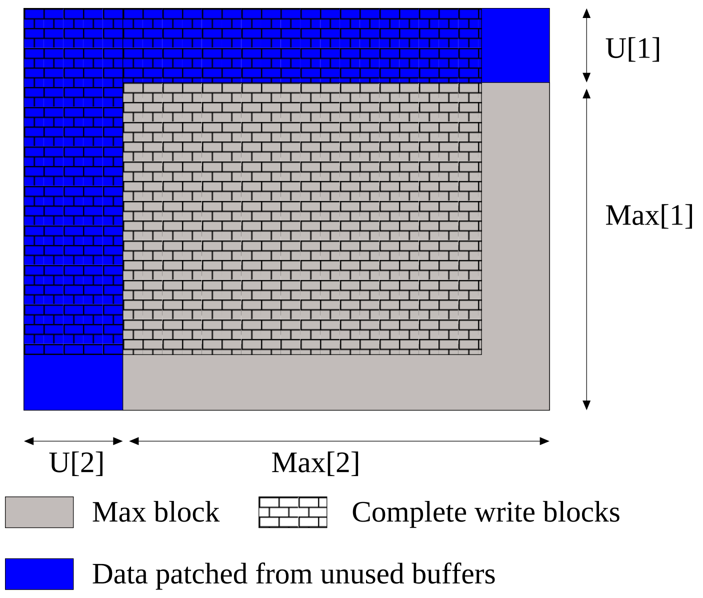


Figure 3.5: In-place construction of complete write blocks in the parallel implementation. The Max block, data from unused buffers, and the constructed write blocks are shown. Note that the regions overlap.

template. Thus additional space for only two templates is required (the space on the output array is not useful in this case as it is used during the transformation and might not have enough contiguous free space to store the intermediates). With the GA/DRA model, using this optimization would involve creating and using DRAs the size of a template.

Parallel Implementation

The parallel implementation is similar to the sequential implementation, whose pseudo-code is shown in Figure 3.9. The Max block and the unused buffers are global arrays. Each Max block is read in directly using the DRA interface. As illustrated in Figure 3.5, the global array corresponding to the Max block is allocated additional space, i.e., dimension *i* of the global array is of size (Max[i]+U[i]). The patching of the data from the unused buffers is done in the additional space allocated in the array such that the complete write blocks form an n-dimensional rectangular region. Thus the construction of the complete write blocks is done in-place, eliminating the movement cost for the data in the intersection between the Max block and the complete write blocks, which do not need to go through the unused buffers. The complete write blocks are then written to disk.

Load Balancing

In the parallel implementation, more than one processor co-operates in performing the transformation. The basic unit of I/O, the Max block, is increased in size to allow all the processors to actively participate in the transformation. In the sequential algorithm, the Max block is defined as the set of read blocks that guarantees that at least one complete write block can be written out in each step. With P I/O processors, the Max block is defined to be the set of read blocks that guarantee that P complete write blocks can be written out in each step.

This set of read blocks can be chosen in a number of ways. To balance the load among the I/O processors, the P write blocks written out in each step should each be handled by a different I/O processor. This allows for a balanced distribution of the I/O load, with all I/O processors actively performing I/O in each step.

A Max block that results in a load-balanced schedule is determined to be a multi-dimensional rectangular region of P write blocks, each handled by a different I/O processor. The read blocks that cover this region form the Max block. A Max block that covers P consecutive write blocks along the fastest varying dimension forms a simple load balanced schedule. However, such a scheme does not take advantage of the flexibility available in choosing the Max block so as to contribute to a global optimal solution. For example, the above scheme would not perform well if the target blocking had a very different orientation. A simple heuristic would be to choose a Max block that aligns with the target block.

In the algorithm design, the Max block was defined first, and other parameters such as memory cost were defined in terms of the Max block. A choice in the Max block determination affects other costs and hence the optimal solution.

An algorithm to enumerate all possible load balanced Max blocks is shown in Algorithm 3.10. Currently, the implementation chooses any load balanced block. The algorithm is based on the observation that the round-robin distribution of the blocks of the disk array enables partitioning of the entire array into load-balanced Max blocks of the same size and shape. If a partition results in the Max block at the origin of the array being load-balanced, all the Max blocks in the partition are guaranteed to be load balanced.

The algorithm can be viewed as a factorization of P to be assigned to different dimensions. The factor assigned to a dimension is the number of write blocks along that dimension to be covered by the Max block. The algorithm represents the array size indirectly using an offset vector. An offset vector is an n-dimensional vector, in which the i-th element represents the distance between two write blocks along that dimension in a linearization of the array into write blocks. For example, for a 10×10 array, blocked using 3×3 tiles, the offset vector is $(1,4)$. The offset is one along the fastest varying dimension. Along the next dimension, it is the number of blocks in all lower dimensions, which is four here. In the algorithm the offset is represented modulo the number of I/O processors. Thus, with two I/O processors the offset vector in the above example is $(1,0)$. In this form, the offset vector also represents the I/O processors that handle the blocks adjacent to the block at origin, along each dimension.

The offset along a dimension can be used to determine the number of different I/O processors that handle blocks if one traverses the array along that dimension. In the above example, if the blocks are identified using a row-column pair, all blocks along the column

```
1: function GenMaxBlocks(blocking,array-dims,P)
2:     Input: array dimensions and blocking, number of I/O processors
3:     Output: Enumeration of load-balanced parallel Max blocks
4:     $D = \Phi$
5:     Compute offsets into "offset[]" array
6:     for each dimension $i$ do
7:         factor$[i] = 1$
8:         if offset$[i] > 0$ then
9:             $D = D + \{i\}$
10:    GenRecursively(P, factor[], offset[], D)
1: function GenRecursively(P, factor[], offset[], D)
2:     Input: #I/O procs, initialized factor & offset arrays, dimensions of interest
3:     Output: Enumeration of load-balanced parallel Max blocks
4:     if P=1 then
5:         /**factor[] has a valid parallel Max block**/
6:         Output factor[]
7:         return
8:     for each dimension $i \in D$ do
9:         if gcd$(P,$offset$[i]) < P$ then
10:            $f = P/$gcd$(P,$offset$[i])$
11:            if $|D| > 1 \vee f = P$ then
12:                factor$[i] = f$
13:                GenRecursively(P/f, factor[], offset[], $D - \{i\}$)
14:                factor$[i] = 1$
```

Algorithm 3.10: Pseudo-code to enumerate all load-balanced parallel Max blocks

$(0,*)$ are handled by I/O processor zero. In fact, an offset of zero along a dimension offset of zero along a dimension implies that all blocks along that dimension are handled by the same I/O processor.

A factor of more than one is assigned to a dimension only if the corresponding blocks chosen are handled by different I/O processors. All dimensions with non-zero offsets are chosen as candidates and are added to the set *D*. Then the routine *GenRecursively* is invoked that recursively determines all load-balanced Max blocks. The routine recursively factorizes *P* and assigns factors to dimensions along the way.

If P has been completely factorized and an invocation of the routine *GenRecursively* finds P to be one, the factorization in factor[] is a load balanced Max block. If not, the routine expands the search along each dimension, by attempting to assign a factor to each dimension and then backtracking to determine more possible solutions.

The possible assignment of a factor to a dimension is determined by $gcd(P, \text{offset}[i])$. The gcd determines the number of different I/O processors that handle blocks along that dimension. If the gcd is 1, it means all the I/O processors own blocks along that dimension. For larger gcds the number of I/O processors is correspondingly lower. The number of I/O processors along a dimension i is given by $f = P/gcd(P, \text{offset}[i])$. Also, along a dimension, all I/O processors own a block before any I/O processor owns a second block. Hence f can be assigned as a factor to that dimension.

3.4.5 Experimental Evaluation

In order to evaluate the effectiveness of the proposed approach, we compared the time for layout transformation using our implementation with the time for transformation using currently available mechanisms. The present interface to DRA is through Global Arrays. When a DRA is to be copied to another DRA with different blocking, the source array is read into a GA one section at a time, and written into the same section of the target array. This is a single-pass solution. The basic unit of access, i.e., the shape and size of the GA needs to be determined. The size is determined independent of the blocking of the source and target arrays to equal the amount of available physical memory. We evaluated three options for the shape of the GA used. One option was to use the largest square tile that fits within the available memory. If the blocking of the DRAs is known, the GA can be chosen to be a multiple of the source block size or the target block size. These three

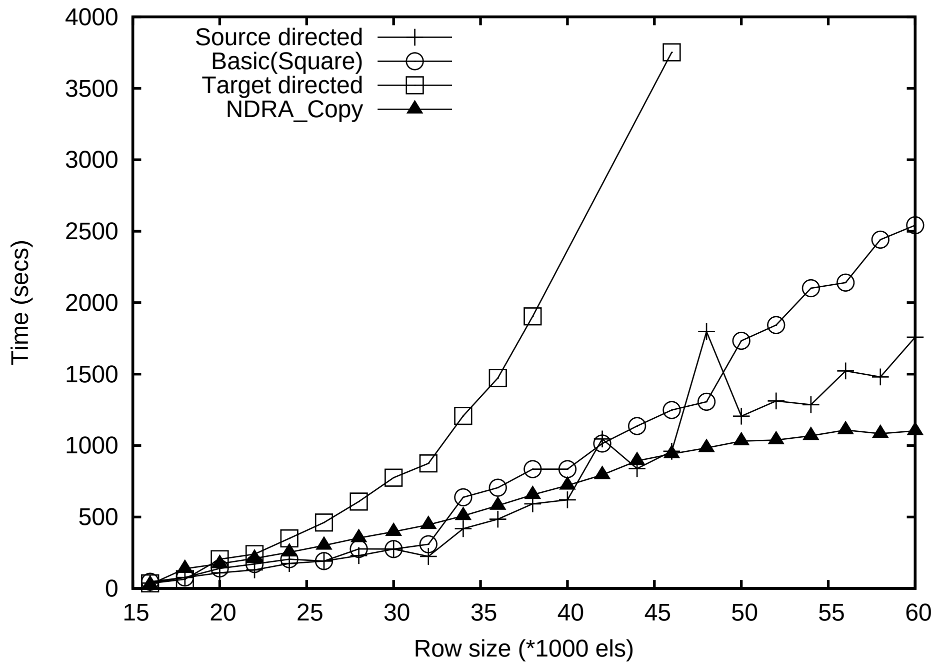


Figure 3.6: Execution time to transform set-of-rows blocking to set-of-columns blocking using a DRA brick size of 1MB

options are labeled Basic(square), Source Directed and Target Directed, respectively. The implementation of the new approach is labeled NDRA_Copy.

We evaluated the mechanisms on the OSCBW machine at the Ohio Supercomputer Center. Each node in the cluster has Dual AMD Athlon MP processors (1.533 GHz) and 2GB of memory. The PGI pgcc 4.0-2 compiler was used to generate the executables. Two sets of experiments were conducted. In one, a set of rows form the blocks in the source array. The target array is blocked as a set of columns. The corresponding results are shown in Figure 3.6. The second experiment involved the reverse – transforming from a set-of-columns blocking into a set-of-rows blocking, and its results are shown in Figure 3.7. The number of rows (or columns) in a block was chosen such that the block size was

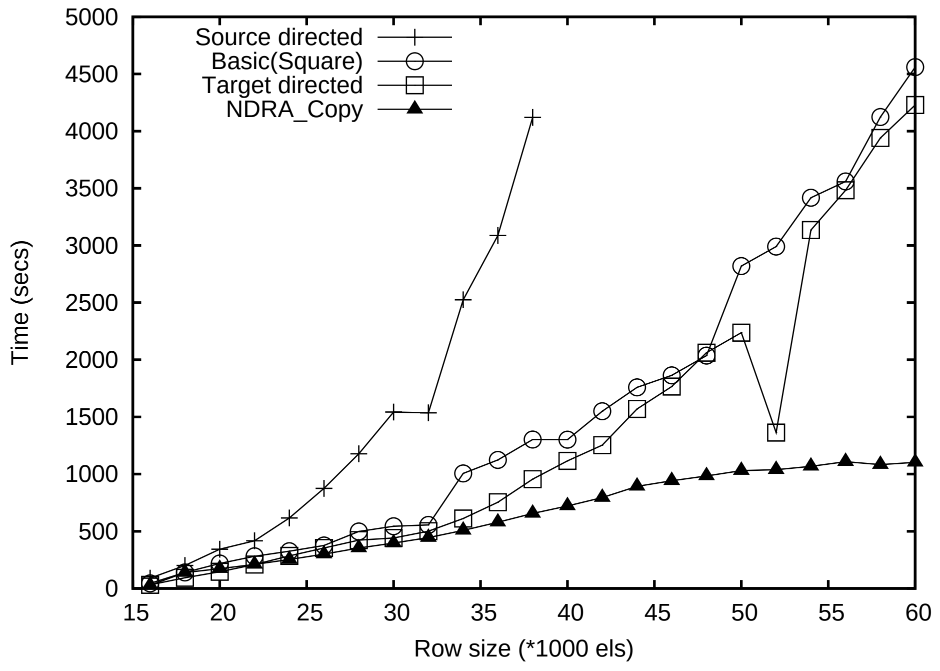


Figure 3.7: Execution time to transform set-of-columns blocking to set-of-rows blocking using a DRA brick size of 1MB

greater than 1MByte, the typical brick size chosen by DRA for this system. For example, for a $\langle 4096, 4096 \rangle$ array, where each element is of size four bytes, set-of-rows blocking corresponds to a block size of 1 MB, with each brick holding a $\langle 64, 4096 \rangle$ block of data; and a set-of-columns layout corresponds to a 1 MB brick holding a $\langle 4096, 64 \rangle$ block of DRA data.

In both the experiments, the array size was increased from 16000 to 60000 in steps of 2000 and all four mechanisms were evaluated. For our approach, the template size is determined automatically using the algorithms described in Section 3.4.4. The x-axis in the graphs shows the array dimension in number of elements. The y-axis shows the transformation time in seconds. We were unable to run larger experiments due to the limited amount of disk space available on the local disks (around 60GB).

In transforming the set-of-rows bricks into a set-of-columns bricks, the target directed method performs significantly worse than other approaches. This is because the data to be read in to memory is not contiguous on disk. The DRA reads in entire blocks of data to 'collect' the data into the global array. This leads to significant increase in cost. Due to this obvious trend, this approach was evaluated with only certain sample array dimensions. The source directed approach performs better, as DRA implementation allows writes of partial blocks, if it is contiguous on disk. Though the unit of write is small, it still performs better than the target directed approach. With larger array dimensions, both the source directed and basic (square) approach increase in cost.

Our implementation performs better than the alternatives. The relative performance benefit of our new approach increases with the size of the array. It starts with a single-pass solution and then uses a two-pass solution for arrays with dimensions larger than 32000. But the execution time increases gradually and is not drastically affected by the exact problem instance at hand. Unlike the other three approaches, our implementation performs comparably for both the transformations evaluated.

The parallel implementation was evaluated on an Itanium 2 cluster at the Ohio Supercomputer Center (OSC)(ia64-osc) and the Mpp2 cluster at the Molecular Sciences Computing Facility in the Pacific Northwest National Laboratory (PNNL) (ia64-pnl). The configuration of the systems is shown in Table 3.8. Initially the data is stored in row-major order on disk. We varied the data access pattern and measured three costs. The *skewed access cost* was first measured for each access pattern. The skewed access cost is the cost of accessing all the elements in the array using the specified access pattern, with the data

	ia64-osc	ia64-pnl
Processor	Dual Itanium 2 (900 MHz)	Dual Itanium 2 (1.5 GHz)
Memory	4GB	8GB
Local disk	80GB	430 GB
Interconnect	Myrinet 2000	Quadrics
Messaging Layer	GM	Elan-4

Table 3.8: Configuration of systems on which matrix reblocking was evaluated

stored in row-major layout. The skew refers to the *misalignment* between the access pattern and the layout of data on disk.

We then measured the cost of transforming the data layout to match the access pattern. This is referred to as the *conversion cost*. Finally, the cost of accessing the elements in the transformed array is measured. The access pattern is now fully aligned with the data layout and this cost if referred to as the the *aligned access cost*.

Table 3.9 shows the costs for a 32768×32768 array of doubles on ia64-osc. The costs were measured on one and two nodes, where one processor was used per node. The costs for a 65536×65536 array on four nodes is shown in Table 3.10. The results for 1, 2 and 4 processors (one per node) on ia64-pnl for a 65536×65536 array is shown in Tables 3.11 and 3.12. Each row in these tables represents a different access pattern being evaluated. The array is accessed in row-major order in units whose size/shape is specified. With P processors, each access corresponds to a read of P such blocks. The size of a block for all the access patterns was 1MB, the size internally chosen by DRA for a brick.

It can be observed that when the access pattern is closely aligned with the data layout on disk, the skewed access cost is higher than the aligned access cost, but not high enough to warrant layout transformation. If the transformed array needs to be accessed multiple

Access Pattern		Access and transformation cost (seconds)					
		#procs = 1			#procs = 2		
Row (#els)	Column (#els)	Skewed access	Conv. cost	Aligned access	Skewed access	Conv. cost	Aligned access
4	32768	176	359	172	97	241	90
8	16384	179	343	178	88	191	88
16	8192	182	345	175	91	173	91
32	4096	196	357	180	105	188	92
64	2048	249	368	181	129	190	93
128	1024	340	372	179	172	202	94
256	512	517	371	183	266	173	93
512	256	861	372	181	434	165	92
1024	128	1580	377	183	749	163	94
2048	64	2994	384	184	1393	167	93
4096	32	5760	373	180	2697	170	95

Table 3.9: Access and transformation cost (in seconds) for a 32768×32768 array stored in row-major order on ia64-osc

Access Pattern		Access and transformation cost (seconds) (#procs=4)		
Row (#els)	Column (#els)	Skewed access	Conv. cost	Aligned access
8	16384	207	733	212
16	8192	238	644	229
32	4096	300	743	230
64	2048	419	723	230
128	1024	650	623	230
256	512	1110	538	230
512	256	2030	466	230

Table 3.10: Access and transformation cost (in seconds) for a 65536×65536 array stored in row-major order on ia64-osc

Access Pattern		Access and transformation cost (seconds)					
		#procs = 1			#procs = 2		
Row (#els)	Column (#els)	Skewed access	Conv. cost	Aligned access	Skewed access	Conv. cost	Aligned access
8	16384	155	370	221	137	249	71
16	8192	209	420	229	177	224	72
32	4096	298	428	292	321	241	69
64	2048	436	423	298	521	265	71
128	1024	734	469	304	973	287	68
256	512	1315	453	307	1938	252	71
512	256	2473	446	316	3648	276	64

Table 3.11: Access and transformation cost (in seconds) for a 65536 $\times$ 65536 array stored in row-major order on ia64-pnl for #procs=1 and #procs=2

Access Pattern		Access and transformation cost (seconds) #procs = 4		
Row (#els)	Column (#els)	Skewed access	Conversion cost	Aligned access
8	16384	54	186	49
16	8192	83	138	63
32	4096	95	133	54
64	2048	129	116	58
128	1024	194	128	62
256	512	322	144	54
512	256	579	149	56

Table 3.12: Access and transformation cost (in seconds) for a 65536 $\times$ 65536 array stored in row-major order on ia64-pnl for #procs=4

times, then the layout transformation cost might be amortized by the lower aligned access cost. As the skew increases, the skewed access cost gets so high as to warrant a layout transformation even if the array is to be accessed just once after the transformation. As expected, the aligned access cost is similar for all block sizes. The layout transformation cost does not vary significantly with the transformation performed. Since I/O is performed in units of an efficient block size determined by DRA, the I/O cost does not vary between transformations unless the number of passes varies. We observe that all the transformations were performed in one pass.

3.5 Related Work

We have used disk I/O volume, coupled with appropriate disk I/O thresholds to optimize out-of-core matrix transposition and reblocking. Two other models for disk I/O have been employed in optimizing out-of-core algorithms. Often the startup time to satisfy is disk I/O request is assumed to dominate the total disk I/O cost. The cost of an out-of-core algorithm is thus measured in terms of the number of I/O operations [37, 57]. In the parallel disk model (PDM) [117], the data is assumed to be organized in terms of blocks. The cost of an out-of-core algorithm is measured in terms of the number of block moves. Several algorithms for sorting and permutation have been proposed based on this model, which have been extended to provide bounds on the number of I/O operations for out-of-core matrix transposition [117, 32]. This model is similar to our cost model, but we focus on the disk I/O volume by allowing the violation of the threshold if it can reduce the number of passes. Schlosser et al. [101] make the interesting observation that multi-dimensional data can be efficiently stored and accessed in current hard disks due to the trends in storage technology. They use this observation to derive a strategy to map data blocks to the linear disk address [103]. This insight can be used to further optimize out-of-core matrix transposition when the layout of the array on disk can be controlled by the application. For more general data layouts, such accurate modeling might not be possible.

We are not aware of any work that minimizes disk I/O volume in reblocking arbitrarily blocked multi-dimensional matrices.

3.6 Conclusion

We addressed the efficient transposition of matrices that are too large to fit in main memory. We formulated the out-of-core matrix transposition problem as an index permutation on the addresses of matrix elements and inferred the effect of various components of the formulation on the I/O time and in-memory permutation time. We discussed the drawbacks of previously proposed algorithms and used empirically derived I/O characteristics of the system to guide the development of our algorithm. We devised an algorithm by choosing the design parameters that minimize the time involved in the I/O and in-memory permutation phases of the algorithm. Thus we improved the overall transposition time, rather than reducing the number of I/O operations, as previous algorithms have done. We subsequently proposed an approach to efficient transformation of the blocked layout of multidimensional disk-resident arrays. The number of passes in the layout transformation is determined based on the specific transformation, such that the overall I/O cost is minimized. The proposed approach was implemented as a new copy primitive within the DRA I/O library.

CHAPTER 4

COMPUTATION MAPPING AND SCHEDULING

4.1 Introduction

Optimizing parallel programs requires effective co-ordination of data movement between the different levels of the memory hierarchy. The computation is scheduled such that the total data movement cost is minimized while maximizing parallelism. In this chapter, we present an approach to automatic management of data movement and scheduling of computation for programs. The data is assumed to be distributed in a global address space in the physical memory or secondary storage associated with the processors. To improve productivity, the framework presents the user with a computation abstraction that allows the expression of locality and parallelism in the computation, organized as a set of independent tasks. This abstraction operates on specific data structures that present data of sufficient granularity for efficient disk I/O and communication. Here, we demonstrate the approach to automatic memory hierarchy management using block-sparse arrays that arise in quantum chemistry calculations such as Coupled Cluster methods [31].

The computation is organized as a set of independent tasks operating on such globally addressable data. Given such a specification, we employ hypergraph partitioning to schedule the computation and the data movement. When the data is distributed amongst the physical memories of processors, we partition the computation amongst the processors while taking the data distribution into account. In the case of out-of-core data structures, we employ hypergraph partitioning to schedule the computation into stages so that each stage can be computed by reading/writing the relevant data elements exactly once. A novel partitioning scheme is proposed to reduce memory consumption within a stage, thus increasing the number of tasks that can be processed within a stage, potentially reducing the disk I/O cost incurred.

4.2 Tensor Contraction Engine

One of the primary motivations for the development of new data abstractions different has been the quantum chemistry models such as Coupled Cluster methods [31]. Tensor Contraction Engine (TCE) [11, 27] synthesis system is a domain-specific compiler for expressing ab initio quantum chemistry models. The TCE takes as input a high-level specification of a computation, expressed as a set of tensor contraction expressions, and transforms it into efficient parallel code. Several compile-time optimizations are incorporated into the TCE: algebraic transformations to minimize operation counts [73, 74], loop fusion to reduce memory requirements [70, 72, 71], space-time trade-off optimization [25], communication minimization [26], and data locality optimization [27, 28] of memory-to-cache traffic.

Each tensor contraction expression is comprised of a collection of multi-dimensional summations of products of several block-sparse input arrays. Consider the following tensor contraction from the domain of quantum chemistry:

$$\begin{array}{l} p1, p2, p3 : O \\ h1, h2, h3 : V \\ i0[p1, p2, h1, h2] \mathrel{+}= -t[p1, p3, h1, h3] * i1[h3, p2, h2, p3] \end{array}$$

where indices $p3$ and $h3$ are contracted out. Here O is the number of occupied orbitals, and V is the number of virtual orbitals. O and V are divided into segments. This segmenting of the dimensions forms a cartesian grid that divides the multi-dimensional array into blocks. An operation on the indices of the segments that form a block determines if that block is non-zero. The sizes of O and V are such that the arrays are too large into fit into the collective physical memory of a parallel system. The arrays are usually stored on the local disks attached to the compute nodes in a cluster, to achieve scalable I/O.

Despite being a variant of matrix-matrix multiplication, the block-sparsity in tensor contractions leads to irregular data access patterns that are not easily tractable. In addition, the difficulty in determining an accurate closed form solution to the size of non-zero data within a tile makes the use of standard out-of-core dense matrix multiplication algorithms a non-trivial task. The wide variation in the sizes of the non-zero blocks, together with the accompanying variation in the data access pattern, makes effective tile-size selection that minimizes the total disk I/O cost a challenging task.

Given such a computation consisting of a set of independent tasks, with each data brick potentially accessed by more than one task, our objective is to determine a schedule for the movement of data bricks between disk and memory, and the processing of the tasks, such that the computation is load-balanced and the total data movement cost is minimized.

4.3 Abstraction for Block-Sparse Matrices

In this section we detail our data abstraction for block-sparse matrices. The abstraction is shown in Algorithm 4.1. For brevity, we use a pseudo-code notation; the actual API is implemented in C/C++. The data abstraction provides collective functions for creating

```
Types::
BsaIndex                                        ▷ Index to block-sparse array
BsaObject                                       ▷ Block-sparse array object

Function Parameters::
BsaObject obj                                   ▷ Handle to block-sparse array
BsaIndex ind                                    ▷ Handle to index to the array
int ndim                                        ▷ Number of dims of the array
int nblocks[ndim]                               ▷ Num blocks along each dimension
int blocks[ndim][nblocks]                       ▷ The size of each block segment
int brick[ndim]                                 ▷ Brick size along each dimension
void *bmap                                      ▷ Bitmap specifying non-zero blocks
Fn_t isNonZero        ▷ Function. Inputs: block indices; Output: true if non-zero block
int *brick                                      ▷ Size of brick along each dim
int *brickIndex[ndim]                           ▷ Index of a brick
void *buf                                       ▷ Local buffer
int dim                                         ▷ Dimension referenced

Functions::
BsaIndex bsaCreateIndex(ndim, nblocks, blocks, isNonZero, brick)
BsaIndex bsaCreateIndex(ndim, nblocks, blocks, bmap, brick)   ▷ Create index – collective
BsaObject bsaCreateArray(ind)                   ▷ Create block-sparse array – collective
void bsaGetBrick(obj, brickIndex, buf)          ▷ Retrieve a brick – one-sided
void bsaPutBrick(obj, brickIndex, buf)          ▷ Store a brick – one-sided
void bsaUpdateBrick(obj, brickIndex, buf)       ▷ Atomically update a brick – one-sided
bool bsaIsBrickNonZero(obj, brickIndex)         ▷ Is a brick is non-zero – one-sided
int bsaGetNBricksAlongDim(obj, dim)             ▷ Enquire #bricks along a dimenion
void bsaDestroyArray(obj)                       ▷ Destroy block-sparse array
void bsaDestroyIndex(ind)                       ▷ Destroy index
```

Algorithm 4.1: Abstraction for multi-dimensional block-sparse arrays

and destroying arrays and non-collective functions to get/put data from/to the distributed block-sparse array.

A brick size is specified while creating a block-sparse array. Alternatively, the user can specify the typical access pattern, to provide hints on the choice of appropriate brick sizes. The non-zero blocks of the array are divided into bricks of this size, which are then distributed amongst the processors in a round-robin fashion. This ensures a uniform distribution of the data among all processors. A small brick size allows for a more uniform distribution of the data amongst the processors. On the other hand, a large brick size allows for coarse-grained, and possibly more efficient, computation and potential reduction in the communication cost, due to amortization of the communication latency.

The process of creating an array is divided into two steps. The index is created first, using function `bsaCreateIndex` from Algorithm 4.1. This involves traversing the bricks in the array, and determining the distribution of the non-zero bricks amongst the processes. The array is then created through function `bsaCreateArray` using this index. The decoupling of the creation of the index from the actual array creation simplifies the construction of multiple aligned arrays using the same index structure. In computations with dynamic allocation and deallocation of memory, the index can be computed once, while the actual memory for the array is dynamically managed.

The arrays can be created by specifying the number of dimensions, the number of blocks, and the actual block sizes. In addition, a bitmap can be provided to specify whether a block is zero. Alternatively, the programmer can provide a function that takes as argument the block indices and returns whether it is zero.

4.4 Computation Abstraction: Task Pool

The task-pool abstraction shown in Algorithm 4.2 enables the specification of a set of independent tasks to be executed in parallel. For each such set, all processes collectively create a `TaskPool` object using the `tpCreateTaskPool` function. All tasks in the task pool take the same number of locality elements as arguments. The access modes for these locality elements can be specified as argument when creating the task pool as well. Three access modes are supported: read, write, and accumulate access modes allow for put, get, and accumulate of global data, respectively.

Each task in the task pool is identified by the routine to be invoked to process that task (accessed through a function handle) and the set of *locality elements* it operates on. In addition, any private data specific to that task can also be specified. Each locality element corresponds to a global data brick, identified by the the brick index, associated with the block-sparse array specified while creating the task pool.

The access mode determines the memory allocation and communication schemes. Each locality element marked `ACCESS_READ` is fetched into a local buffer before starting the computation. It might also be cached for future references. Any locality elements marked `ACCESS_WRITE` or `ACCESS_UPDATE` involve communication after completion of the task. The computation partitioning and the mapping strategy determine if these locality elements can be cached. The routine specified by the user to process the task takes as input local buffers containing the relevant data. The framework is responsible for handling the data movement and providing these buffers to the user's routine.

Tasks are added to a task pool using the `tpAddTask` function. The creation and addition of tasks to the task pool is done by all the processes, in a replicated fashion. Once all the tasks have been added to the task pool, `tpSeal` is used to *seal* the work pool. This function is invoked once for a task pool and is used to perform start-time optimizations.

Subsequently, all the processes collectively invoke `tpProcess` to process the tasks in the task pool. A task pool, once created, can be processed multiple times. The cost of start-time optimizations, performed once, are thus amortized.

The work-sharing construct is illustrated using an implementation of block-sparse matrix multiply, shown in Algorithm 4.3. The multiplication is of the form

```
1:  Types::
2:  TaskPool                                          ▷ Handle to task pool
3:  Fn_t                                              ▷ Function to process a task
4:  LocalityInfo                                      ▷ Locality information
5:  PrivateData                                       ▷ Local information for a task
6:  AccessMode                                        ▷ Mode of access to arguments of each task
7:  BsaObject                                         ▷ Block-sparse array object
8:
9:  Function Parameters::
10: TaskPool tpHandle                                 ▷ Task pool
11: Fn_t fn                                           ▷ Processing function
12: Integer nLocInfo                                  ▷ #locality elements
13: LocalityInfo *locs                                ▷ Locality elements
14: PrivateData *pvt                                  ▷ Local data
15: AccessMode *modes                                 ▷ Access mode for each locality element
16: BsaObject *objs                                   ▷ Block-sparse arrays operated on
17:
18: Functions::
19: TaskPool tpCreateTaskPool(objs, modes)            ▷ Create task pool
20: void tpAddTask(tpHandle, fn, nLocInfo, locs, pvt) ▷ Add task
21: void tpSeal(tpHandle)                             ▷ Seal task pool
22: void tpProcess(tpHandle)                          ▷ Process task pool
```

Algorithm 4.2: Task pool abstraction

$$C[i,j] += A[i,k] * B[k,j]$$

The brick sizes along the different dimensions are assumed to be defined elsewhere. Parameters `bsaA`, `bsaB`, and `bsaC` correspond to block-sparse arrays A, B, and C, respectively.

Algorithm 4.3 shows routine `BrickMatmul` used to process an individual task, a matrix-matrix multiplication involving a brick from each of the arrays. Note that no explicit communication is involved. The routine assumes that all input data are read into local memory and all output data are written/accumulated into global memory. Algorithm 4.3 also shows the implementation of parallel matrix multiplication using this routine.

4.5 Hypergraph Partitioning Problem

A hypergraph is a generalization of an undirected graph in which an edge, referred to as a *net*, can connect more than two vertices. The hypergraph partitioning problem is concerned with dividing a hypergraph into a set of P sub-hypergraphs, for a given P, such that the cost of interconnection between the parts is minimized. The cost is influenced by the nets shared between more than one part, with a variety of metrics defined on them. The principal idea behind the definition of the objective function is to minimize the cost incurred by assigning related entities, represented by vertices connected by a net, to distinct parts. In the rest of the section, we shall present a formal description of the hypergraph partitioning problem and define relevant cost metrics.

A hypergraph $H = (V,N)$ is defined as a set of vertices V and a set of nets (hyper-edges) N among those vertices. Each net $n_j \in N$ is a set of vertices from V. Weights (w_i) and costs (c_j) can be assigned to the vertices $(v_i \in V)$ and edges $(n_j \in N)$ of the hypergraph, respectively. $\Pi = \{V_1, V_2, \ldots, V_P\}$ is a P-way partition of H if (1) each part V_i is a non-empty subset of V, (2) the parts are pairwise disjoint, and (3) union of the P parts is equal to V. A partition is said to be *vertex-weight-balanced* if

$$W_p \leq W_{avg}(1+\varepsilon) \text{ for } 1 \leq p \leq P$$

where $W_p = \sum_{v_i \in V_p} w_i$ is the sum of the vertex weights of part V_p, $W_{avg} = (\sum_{v_i \in V} w_i)/P$ is the weight of each part under the perfect load balance condition, and ε is a predetermined maximum imbalance ratio allowed.

In a partition Π of H, a net that has at least one vertex in a part is said to connect that part. The *connectivity* λ_j of a net n_j denotes the number of parts connected by n_j. A net

n_j is said to be a *cut* if it connects more than one part (i.e., $\lambda_j > 1$). The cut nets are also referred to as external nets, and their set is denoted by N_E.

A P-way partition Π of H can also be viewed as inducing $(P+1)$-way net partitioning, with P internal net sets and one external net set N_E; that is, $\Pi = \{N_1, N_2, \ldots, N_P, N_E\}$. Here for all internal nets $n_j \in N_p$, all the vertices of those nets belong to the same part, i.e., $n_j \subseteq V_p$ for $1 \leq p \leq P$. Similarly to a vertex-weight-balance partition, a partition is said to be *net-cost-balanced* if

$$C_p \leq C_{avg}(1+\varepsilon) \text{ for } 1 \leq p \leq P$$

where $C_p = \sum_{n_j \in N_p} c_j$ is the sum of the internal net costs of part p, and $C_{avg} = (\sum_{n_j \in N-N_E} c_j)/P$ denotes the average internal net cost under the perfect load balance condition.

There are various ways of defining the cut-size $\chi(\Pi)$ of a partition Π [76]. The two relevant ones for our context are cut-net and connectivity-1, defined as follows:

$$\chi(\Pi) = \sum_{n_j \in N_E} c_j \tag{4.1}$$

$$\chi(\Pi) = \sum_{n_j \in N_E} c_j(\lambda_j - 1) \tag{4.2}$$

With the cut-net metric (4.1), each cut net n_j contributes its cost to the cut, whereas with the connectivity-1 metric (4.2), each cut net n_j contributes $c_j(\lambda_j - 1)$ to the cut-size. The hypergraph partitioning problem can be defined as the task of dividing a hypergraph into two or more parts such that the cut-size is minimized, while a given balance criterion either among the part weights or net costs is maintained. Algorithms based on the multi-level paradigm, such as hMETIS [55] and PaToH [115], have been shown to compute good partitions quickly for this NP-hard problem.

4.6 Optimizing Computations on In-Memory Data

4.6.1 Problem Definition

A computation is to be performed on globally addressable data. The data is partitioned into non-overlapping regions and is distributed across the memories of the processors, such that each region is assigned to one and only one processor. The computation is expressible as a set of independent tasks. Each task takes as input a set of data regions and reads, writes, and/or updates (accumulates) one or more data regions. The computation cost of each task is also provided.

Note that each task can be executed on any processor. The input data regions associated with the task are brought into local memory and the task is executed. The output data are then written/accumulated into the global regions. If a task is executed on a processor that contains the data regions required by it, no communication is required. In addition, if a set of tasks that require the same data regions are co-located in a processor, communication cost can be significantly reduced by reusing the read-only data across tasks.

We assume that we have enough memory to store all the data required by all the tasks. Thus, given a set of tasks assigned to a processor, the amount of communication performed by that processor is equal to the total size of all the distinct data regions accessed by the set of tasks assigned to it.

The objective is to partition the set of tasks among the available processors such that the amount of communication required is minimized, while maintaining the balance of computational load amongst the processors.

4.6.2 Communication Minimization: Locality-Aware Load-balancing

We model the problem of locality-aware load-balancing as a hypergraph partitioning problem. Each data region and task in the computation has a corresponding vertex in the hypergraph. A net is introduced in the hypergraph for every data region in the computation. For each data region, the corresponding net connects the vertices corresponding to it and the tasks that access it. The weight associated with each net is the communication cost associated with the data region. We model it to be the size of the data region. The cost of a vertex is zero if it corresponds to a data region, and is the number of operations to be executed if it corresponds to a task.

We can evaluate the hypergraph thus constructed in two ways. It can be used to determine the assignment of both the tasks and data regions to processors. If the data regions are pre-distributed and cannot be remapped, the distribution of the data regions amongst the processors can be pre-specified by constraining each data region to be on a specific processor. The hypergraph is then partitioned to determine the mapping of the tasks to the processors. Given a partition, the cost incurred by a net is the size of the corresponding data region times the number of remote processors that have been assigned at least one task that accesses this region. The total cost of all the nets is given by the connectivity metric, shown in equation 4.2.

4.6.3 Experimental Evaluation

We evaluated the primitives by comparing them with alternative schemes on the Colony2a system in Pacific Northwest National Laboratory. It is a twenty-four node cluster with each node being a dual 1GHz Itanium-2 with 6GB memory. We used the Infiniband network available on the cluster for our experiments.

Three alternative load-balancing schemes were implemented for comparison. In the first scheme, henceforth referred to as the *Random* scheme, each processor traverses the entire list of tasks in the same order. For each task in the traversal, each processor generates a pseudo-random number between 0 and $P-1$, where P is the number of processors. If the random number generated is the processor's rank, the processor executes that task. Since all the processors start with the same random seed, they all generate the same sequence of pseudo-random numbers. This ensures that each task is executed by exactly one process. The randomization results in a uniform distribution of the number and sizes tasks to processors. Note that this scheme balances the number of tasks and not task execution times. In addition, locality is not taken into account.

In the second scheme, one of the locality elements in each task is marked. Each task is executed by the process that "owns" the marked locality element in that task. This scheme is referred to as the *Owner* scheme. This scheme ensures locality for the array used to determine the ownership. Though the round-robin distribution ensures a reasonably balanced distribution of the data and hence the ownership, computational load is not guaranteed to be balanced.

The third scheme is based on dynamic load balancing. In this scheme, referred to as *NextTask*, all the processes enumerates the tasks to be executed in the same order. A global shared counter is used to determine the next task to be executed. Each process, when idle, performs an atomic *fetch-and-add* of the global shared counter. The value obtained by the process specifies the next task to be executed by it. All processes continue this procedure until the counter exceeds the number of tasks to be processed. The strictly increasing counter ensures that no task is executed more than once. It also keeps all the processes busy, till there are no more tasks to be executed. This ensures load balancing. But locality is not taken into account.

Note that this scheme is similar to self-scheduling in OpenMP [88]. This is also the typical model of parallelization used in many applications, including some quantum chemistry codes [48].

Execution times of the following tensor contraction expression, typical of those encountered in quantum chemistry, were measured:

$$
\begin{aligned}
& a,b,c,d : O \\
& i : V \\
& C[a,b,c,d] = A[a,b,i] * B[i,c,d]
\end{aligned}
$$

where O and V correspond to the number of occupied and virtual orbitals, respectively. They are divided into a number of symmetry segments, in turn dividing the matrix into a set of blocks. For example, if O is divided into four symmetry segments, array C would consist of 64 blocks. A block of a matrix is non-zero if a function of its block segment indices is equal to the symmetry value associated with the matrix. Typically, the function is an exclusive OR operator and the symmetry of a matrix is zero. The tensor contraction is, in effect, a block-sparse matrix multiply. The indices were divided into four symmetry segments. The O index was set at 160 with four symmetry segments of length 80, 40, 20, and 20, respectively. The value of V was varied to be a multiple k of O, with k varying from 1 to 16. The number of processors was varied from 2 to 32.

The execution times are shown in Figure 4.1. The three alternative schemes, labeled *Random*, *Owner*, and *NextTask*, and our approach, labeled *Our*, are shown. For our approach the cost is shown including and excluding the overhead of hypergraph partitioning.

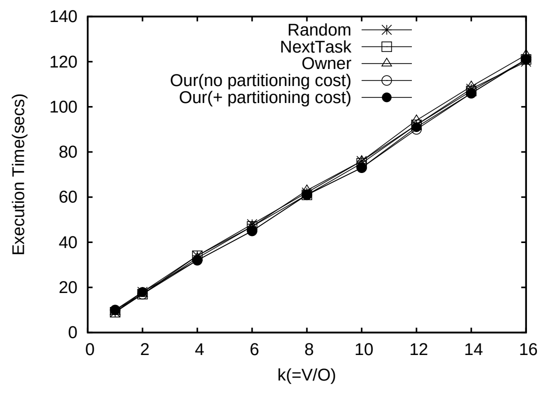
For smaller numbers of processors, the communication cost and the hypergraph partitioning overhead are not significant. Hence, the difference in the performance of the various schemes is minimal. Increase in the number of processors increases the communication cost. Our scheme, being locality-aware, performs increasingly better than the other schemes. The NextTask scheme, which completely ignores locality, performs progressively worse. The Owner scheme ensures locality for at least one of the arrays, thus performing better. The Random scheme, performs better than the NextTask and Owner schemes, due to the benefits of randomization.

The cost of hypergraph partitioning increases the cost of our load-balancing mechanism. Though the overhead increases with increase in the number of processors, our mechanism still performs better, even when partitioning overhead is taken into account. Note that in typical applications, the partitioning overhead is amortized over multiple processings of the same task pool.

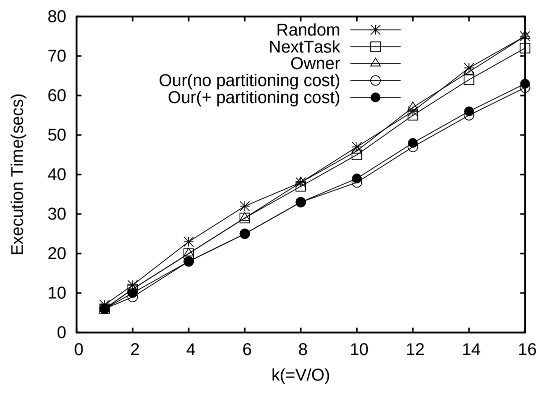
The speedups obtained by the different schemes, for k value being 8 and 16, are shown in Figure 4.2. The sequential execution times were determined for these problem sizes, and are shown in the figure. Our approach achieves a speed-up of up to 20 on 32 processors, excluding partitioning cost. Note that we employ blocking communication.

4.7 Optimizing Computations on out-of-core data

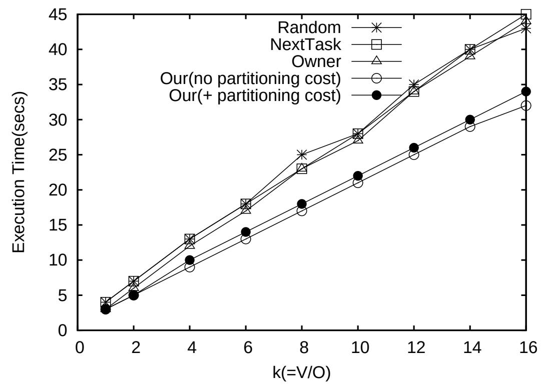
In this section, we present a novel application of hypergraph partitioning to automatically determine the computation and I/O schedule. We begin with a definition of the problem and a discussion on the limitations of a direct application of the hypergraph partitioning model. We then present an alternative formulation that better solves our problem of interest.



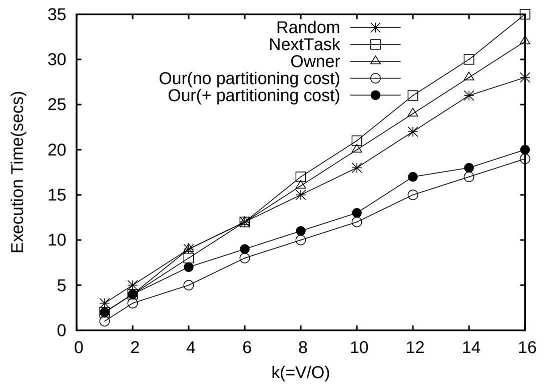
(a) 2 processors



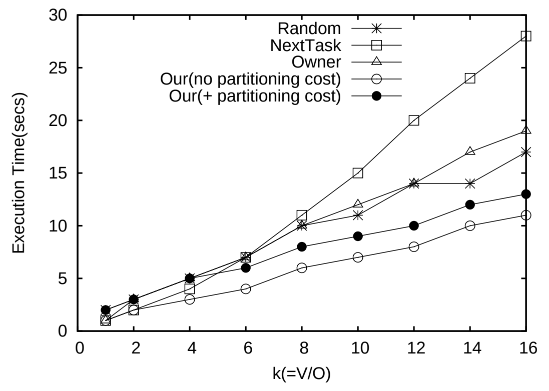
(b) 4 processors



(c) 8 processors

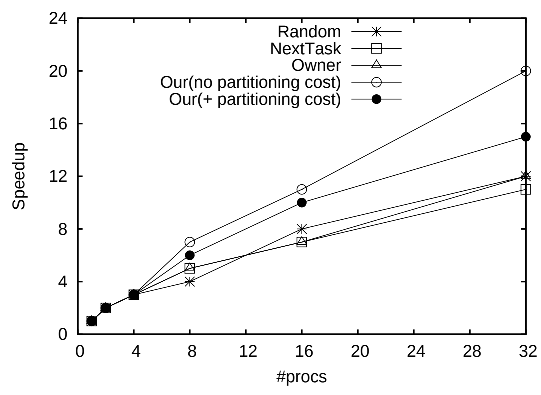


(d) 16 processors

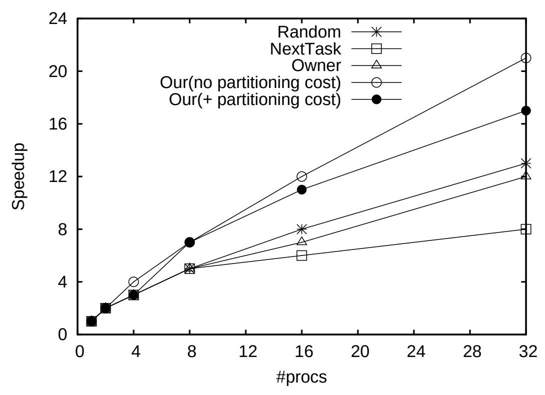


(e) 32 processors

Figure 4.1: Execution time, in seconds, of block-sparse matrix multiply for various schemes. Time is shown in y-axis. k=(V/O) is shown along x-axis. Each graph corresponds to a different number of processors.



(a) k=8 (sequential time=111 secs)



(b) k=16 (sequential time=219 secs)

Figure 4.2: Scalability of the schemes for block-sparse matrix multiply for (a) k=8 and (b) k=16. The number of processors is shown in x-axis. Speedup is shown in y-axis. The corresponding sequential execution times are also shown.

4.7.1 Problem Definition

We are given a computation consisting of a set of independent tasks, with each task accessing a set of data elements. The data elements are in secondary storage and each data element is potentially accessed by more than one task.

The objective is to determine a computation schedule, so as to minimize the total disk I/O cost. The schedule for a computation consists of a sequence constructed from the following five operations:

Read Read a brick into physical memory

Write Write a brick to disk

Allocate Allocate memory for a brick

Deallocate Free memory allocated to a brick

Compute Process a task

Note that buffers for all bricks need to be allocated and de-allocated, whereas disk I/O schedule is to be determined only for the input and output bricks. The schedule is required to ensure that at any point in the processing of the tasks, the total memory allocated to the data bricks is less than the memory available. In the case of a parallel system, the global memory available is the constraint imposed on the I/O schedule.

4.7.2 Disk I/O Minimization: One-Level Partitioning

In this section, we describe a direct application of hypergraph partitioning to the disk I/O minimization problem. The construction of the hypergraph is described, followed by the partitioning procedure to derive a valid computation and I/O schedule.

A *task-brick* hypergraph is constructed from the set of tasks and the set of data bricks accessed by them. For each task and data brick, a vertex and a net is added to the hypergraph, respectively. For each data brick, a net is constructed that connects the vertices corresponding to the tasks that access that brick. The cost associated with the net corresponds to the data movement cost associated with the data corresponding brick. We model this cost to be the size of the brick. The weight associated with each vertex is proportional to the computation cost associated with the corresponding task. In the evaluation of our scheme, this is specified to be number of operations involved.

Common applications of hypergraph partitioning deal with parallelization, and hence have a pre-specified number of parts into which the hypergraph needs to be partitioned. We are interested in partitioning the computation into stages such that the memory requirement at any point in the computation does not exceed the memory available.

We model this problem using hypergraph partitioning together with the memory usage constraint. We recursively partition the given hypergraph into two stages when the computation represented by it cannot be executed without violating the memory constraint. The memory usage of a part is determined as the sum of weights of all nets incident or internal to the corresponding sub-hypergraph. We shall refer to the solution thus obtained as the one-level partition.

Figure 4.3 illustrates a one-level partition of a task-brick hypergraph. The computation involves nine tasks and six data elements. The figure shows the tasks as squares and the data elements as nets (set of edges connected by circles.) All data elements are assumed to be of the same size. Let the memory in the system be large enough to hold three data elements. The partitioning of the hypergraph into three stages, indicated by the three enclosing rectangles, is shown. Each partition requires three data elements to complete processing. Two of the nets, labeled n_1 and n_2, are cut-nets and are accessed in more than one stage. For each cut-net, dummy vertices are introduced in each partition on which it is incident, to represent its contribution to the memory cost of that partition. The total I/O cost is 9 data elements, the number of data elements within each part in the partition.

Given such a partition, the computation schedule is shown in Algorithm 4.4. The schedule corresponds to reading all input bricks relevant to a part, computing the relevant tasks and writing out any output bricks back to disk. In a parallel system the processing of tasks is done using a simple load-balancing mechanism in which each idle process chooses the next task to execute from an total order of all tasks to be executed in the current stage. There is no reuse of data across the different stages. Thus, a reduction in the number of stages is generally beneficial. The algorithm also shows the schedules for memory allocation and deallocation.

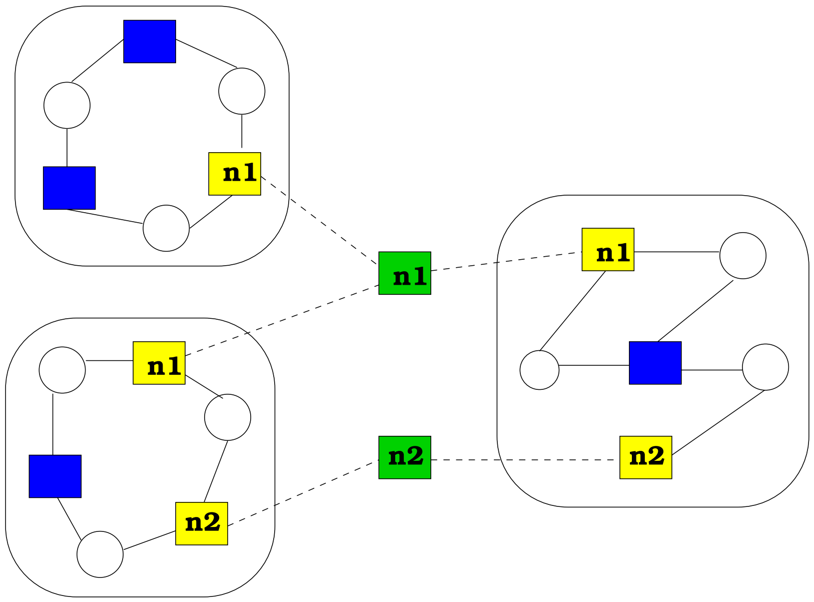


Figure 4.3: Illustration of one-level partitioning

4.7.3 Read-Once Partitioning

The above approach is simplistic in the measurement of the memory cost for each stage. It ignores the potential for reuse across the stages. In addition, the reuse is determined to be between all the tasks in a given stage. While hypergraph partitioning improves the data reuse within a stage, the available memory can be better utilized by further investigating the reuse relationships between the tasks in a stage. This would enable the scheduling of computation and disk I/O so that only a subset of the data elements in a given stage need to be allocated memory at any moment. This improves the memory utilization and potentially reduces the disk I/O cost.

We present an alternative use of hypergraph partitioning that achieves this. We shall refer to such a partitioning as *read-once partitioning*.

A read-once partition is a partition of a task-brick hypergraph such that the sum of the sizes of the cut-nets, corresponding to data bricks accessed in more than one part, and the size of data uniquely accessed in any part does not exceed the available memory. This partition induces a schedule in which the processing of tasks is organized into steps, one for each part in the partition. The processing is preceded by moving all data elements accessed by more than one step, referred to as shared bricks, into memory. Each step is processed by first allocating memory for data elements local to that step and performing the necessary disk I/O. The tasks in the current step are then processed and the updated bricks local to this step are written back to disk. The memory allocated for the local bricks are finally reclaimed. The procedure is then repeated for the next step. After processing all the steps, any updated shared bricks are written to disk. The computation schedule for a read-once partition is shown in Algorithm 4.5.

Thus a set of tasks, while requiring data elements that together cannot fit in the memory available, can potentially be scheduled to be processed using the available memory. By keeping all cut-nets in memory throughout the computation of the given set of tasks, this approach also avoids redundant I/O for any accessed data element.

The scheme uses a pessimistic upper-bound in its calculation of the memory cost due to the allocation of all cut-nets at once, even though a cut-net might be used only much later. Despite this apparent inaccuracy, this scheme significantly improves memory utilization by deallocating nets internal to a step once they are used, thus allowing more related tasks to be processed within a stage.

Note that the number of parts (steps) in a read-once partition is not significant, as increasing the number of parts does not increase the disk I/O cost. But choosing an arbitrarily large number of parts can distribute related tasks, increasing the total size of the cut-nets, thus making a read-once partition infeasible. We choose a simple scheme of a linear search for the number of parts, starting from two. For each choice of the number of parts, a net-cost-balanced hypergraph partitioning with cut-net metric is computed, and the result is checked to be a feasible read-once partition (i.e., $\mathit{cutsize} + C_p \leq \mathit{memory\_limit}$ for $1 \leq p \leq P$). If it is not, we continue the search for a read-once partition by increasing the number of parts. In the current implementation, we limit the number of parts being searched to be less than 128, which we found to be sufficiently large in practice.

4.7.4 Integrated Approach: Two-Level Partitioning

The integrated algorithm, `TwoLevel`, is shown in Algorithm 4.6. It returns a set of ordered pairs, each pair specifying the set of tasks in that stage and the computation schedule obtained using read-once partitioning. If a read-once partition exists that satisfies the memory constraint, using the procedure `ReadOnce`, the set of tasks together with the computation schedule is returned. If not, the algorithm proceeds recursively by partitioning the set of tasks to balance the net-weights, using the `NetBalancePartition` routine, and solving the two parts independently and combining the result.

The outer-level partitioning scheme is identical to that used in one-level partitioning. They differ primarily in mechanism used to decide whether a part (sub-hypergraph) needs to be further partitioned.

Figure 4.4 shows a possible partitioning of the same computation as in Figure 4.3 using the two-level approach. The stages in the computation, corresponding to the parts in the outer-level partition are indicated by enclosing rectangles. Enclosing circles are used to show the parts in the read-once partitions within each stage. Nets n_1 and n_2 are the cut-nets, similar to the partition determined in Figure 4.3. Two of the stages produced by the

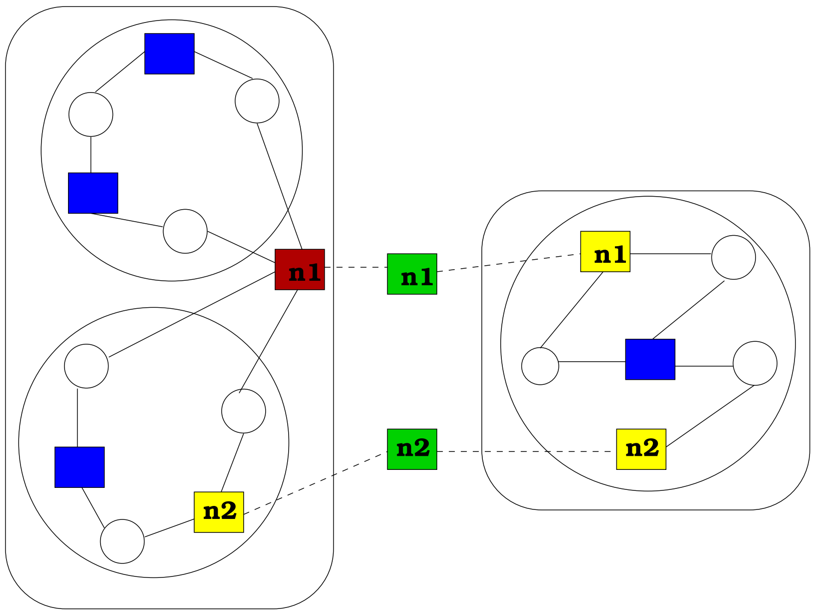


Figure 4.4: Illustration of two-level partitioning

one-level partitioning approach now form the two steps of a read-once partition in a single stage. Net n_1 is a cut-net for that read-once partition and is retained in memory through the processing of both the steps in the stage. This is indicated by the single representative vertex for n_1 in that stage being shared by both the steps. The memory constraint is still satisfied as the memory usage does not exceed the size of three data elements at any point. The total disk I/O cost for this partitioning is equal to the size of eight data elements, as compared to nine for the partitioning in Figure 4.3.

Note that the illustration shows only one possible partitioning and there maybe many equivalent partitions. Also, unlike in the illustration, the partitions produced by the two-level partitioning approach need not, in general, correspond to any one-level partitioning that is the best possible for the given hypergraph.

Definition 2. *A valid part in a one-level partition is defined as a sub-hypergraph of the task-brick hypergraph such that the sum of weights of its incident and internal nets does not exceed the memory available.*

Lemma 1. *A valid part p in a one-level partition corresponds to a trivial read-once partition.*

Proof. Since p is a valid part in a one-level partition, the sum of nets accessed by the corresponding sub-hypergraph satisfies the memory constraint. Given such a sub-hypergraph, the read-once partitioning algorithm can construct a trivial read-once partition with only one part. All nets accessed in that partition are global to the read-once partition, with no nets being local to the only part. □

Lemma 2. *Barring the termination condition, both the algorithms form the same recursive bisection trees.*

Proof. Both use the same partitioning algorithm to divide a hypergraph into two sub-hypergraphs. Since both procedures recursively partition a given hypergraph into two parts, they form identical recursion trees in which each node corresponds to a hypergraph that is partitioned into its children. □

Lemma 3. *The sub-hypergraphs encountered in the recursive procedure for the two-level partition are a subset of the sub-hypergraphs encountered in the recursive procedure for the one-level partition.*

Proof. From Lemma 2, the recursion trees of both the algorithms are identical, barring the termination condition. From Lemma 1, when the one-level approach determines a valid part and stops further partitioning, the two-level approach determines a read-once partition and stops as well. Note that the two-level partition might determine a sub-hypergraph encountered in the recursion procedure to be a valid read-once partition and stop further refinement, while it might not be a valid part in a one-level partition. This might lead to further refinement being required in the one-level approach than the two-level approach. $\square$

Theorem 1. *The solution obtained by two-level partitioning is no worse than that obtained by one-level partitioning.*

Proof. From Lemma 3, only a subset of the recursion tree from the one-level approach is encountered in the two-level approach. Thus, there is no new or different partitioning in the two-level scheme as compared to the one-level scheme. Since only partitioning can increase the disk I/O cost, the I/O cost for the two-level approach is no worse than that for the one-level approach. $\square$

4.7.5 Experimental Evaluation

In the experimental evaluation, we will focus on evaluating the two-level partitioning scheme. We evaluate our approach using the following Coupled Cluster Doubles (CCD) [31] sub-calculation:

$$
\begin{array}{l}
p3, p4, p5, p7 : V \\
h1, h2, h6, h8 : O \\
\text{input-output arrays} : i0, t, v1, v2 \\
\text{intermediate arrays} : i1 \\
i1[h6, p3, h1, p5] += v1[h6, p3, h1, p5] \\
i1[h6, p3, h1, p5] += t[p3, p7, h1, h8] * v2[h6, h8, p5, p7] \\
i0[p3, p4, h1, h2] += t[p3, p5, h1, h6] * i1[h6, p4, h2, p5]
\end{array}
$$

O is set to have four segments (40,40,20,20), and V is divided into the four segments (100,100,60,60). The input/output arrays are assumed to be created and passed as inputs to the execution environment. The first operation initializes the intermediate array. The subsequent arrays produce and consume the intermediate. The initialization operation is implemented in a data-parallel fashion with each process initializing the data bricks local to it.

We evaluate our approach, henceforth also referred to as *HpGraph*, by comparing it with the approach taken in state-of-the-art quantum chemistry packages such as NWChem [47]. In this scheme, the data elements, stored in a bricked form, are replicated across the local disks of the processors. A simple load-balancing scheme is used to distribute the computation amongst the processors. Each process chooses the next brick of the output array to be computed, in a linear ordering of the non-zero bricks, and proceeds to process it by fetching the required bricks from the input arrays and computing the partial products. The computation is performed by transforming the data layouts to ensure contiguity of the contracted indices, following an invocation of DGEMM. The resulting output brick is then written to the replicated copy of the array on the local disk. Before the output array can be used as an input in another tensor contraction, the local modifications to the replicated array need to be *reconciled*. This is essentially an accumulation operation in which all partial contributions to the individual bricks are added together in an operation similar to MPI_AllReduce. This scheme was implemented using our data abstraction, with suitable extensions to replicate and reconciles disk arrays.

This alternative scheme is similar to the *NextTask* scheme introduced in Section 4.6.3 and will be labeled as such. The inputs are assumed to be replicated while evaluated this scheme. A reconcile operation is carried out on $i1$ before it is consumed to contribute to $i0$. In addition, the output array $i0$ is reconciled at the end. All inputs are assumed to be distributed while evaluating our scheme, and no cost is incurred in reconciling any of the arrays.

The memory limit for our scheme was set to 1 GB on each of the systems. While under-utilizing the memory increases the overall cost of the computation, the results show efficient utilization of even a portion of the memory leads to significant improvements. In addition, the un-utilized memory can be used for optimizations such as a caching to further reduce the communication cost. Note that utilizing the entire memory for the computation might degrade performance due to interference with the operation of the operating system and the disk buffer cache.

We evaluated the two schemes on the following three systems:

ia64-osc A cluster with dual Itanium-2 900MHz nodes, each with 4GB physical memory, and 80GB local disk, and a Myrinet 2000 interface. GM is the underlying communication protocol.

ia64-pnl A cluster with dual 1GHz Itanium-2 nodes, each with 6GB physical memory, 80GB hard drive and GM interconnection network.

p4-osc A cluster with each node containing two 2.4GHz Pentium 4 processors and 4GB physical memory, 80GB local disk, and an Infiniband interconnection network.

The sub-calculation was evaluated on the three systems by varying the number of nodes between 1 and 8. Note that only one CPU in each node was utilized in all three clusters.

The average disk I/O costs per process for ia64-osc, ia64-pnl, and p4-osc are shown in Figs. 4.5, 4.6, and 4.7, respectively. On ia64-osc and p4-osc, the effective orchestration of the data movement leads to a reduction in the disk I/O cost even in the sequential case. The improvement over the alternative scheme increases with the number of processors, achieving a factor of 11 on p4-osc for 8 processors. We believe the worsening disk I/O cost for two processors for NextTask on ia64-osc is due to an ineffective task distribution that

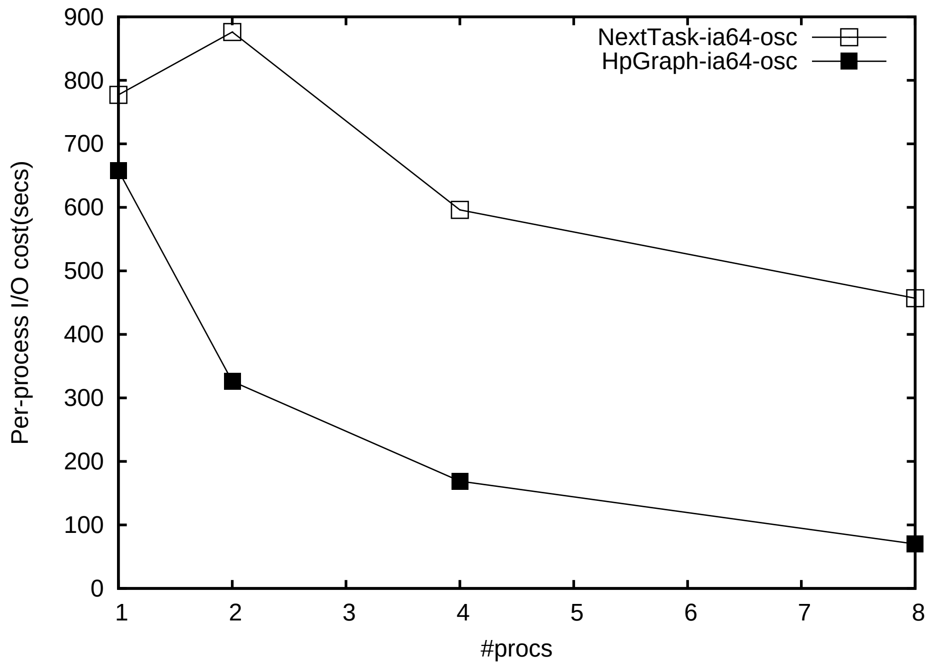


Figure 4.5: Average per-process I/O cost, in seconds, on ia64-osc

results in both processes accessing most of the data bricks, while the data access pattern increases the miss rate on the system buffer cache for disk I/O.

The sequential disk I/O cost of HpGraph is observed to be worse than NextTask on ia64-pnl. We believe this is due to the increased memory size that supports a larger system buffer cache, resulting in an improved reuse for the alternative approach. But an increase in the number of processors leads to performance trends similar to those on the two systems.

The turnaround times are shown in Table 4.1. In addition to improving the disk I/O cost, the turnaround times for HpGraph, including the cost of hypergraph partitioning, are consistently better than that for NextTask. On p4-osc for eight processors, HpGraph leads to a 49% improvement over NextTask, with similar trends observed for other processes.

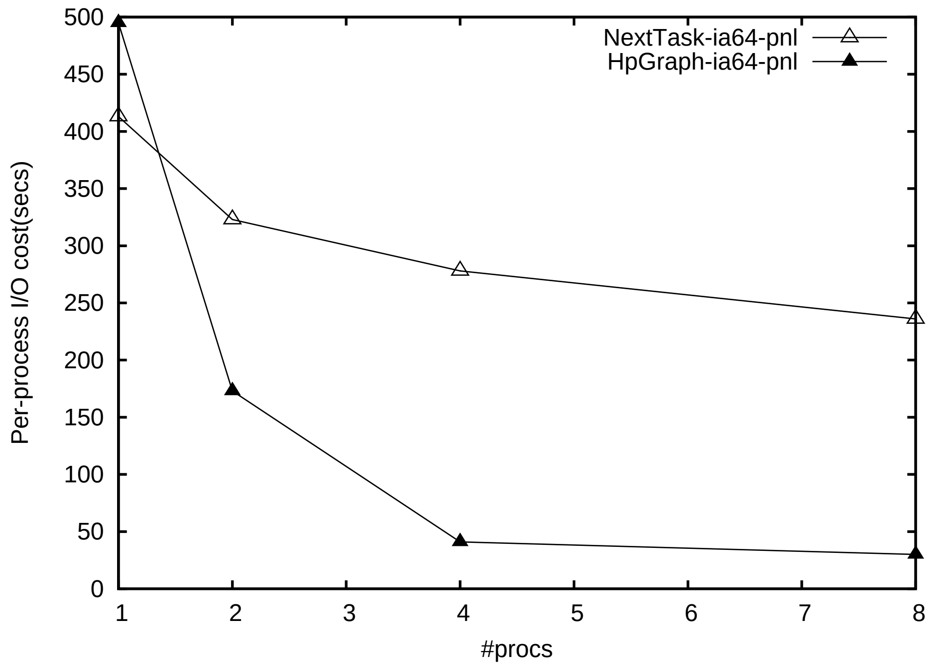


Figure 4.6: Average per-process I/O cost, in seconds, on ia64-pnl

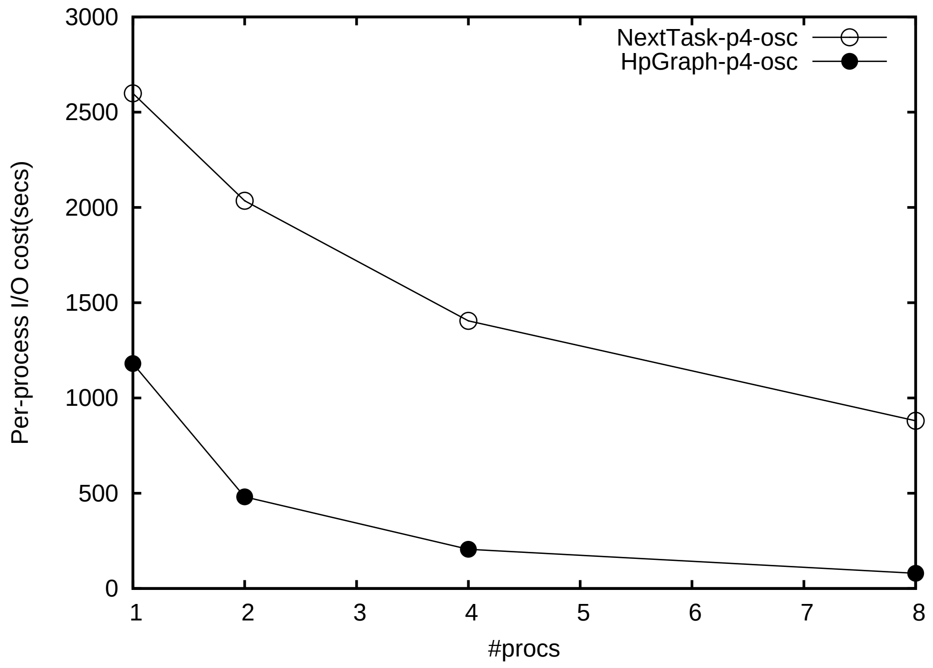


Figure 4.7: Average per-process I/O cost, in seconds, on p4-osc

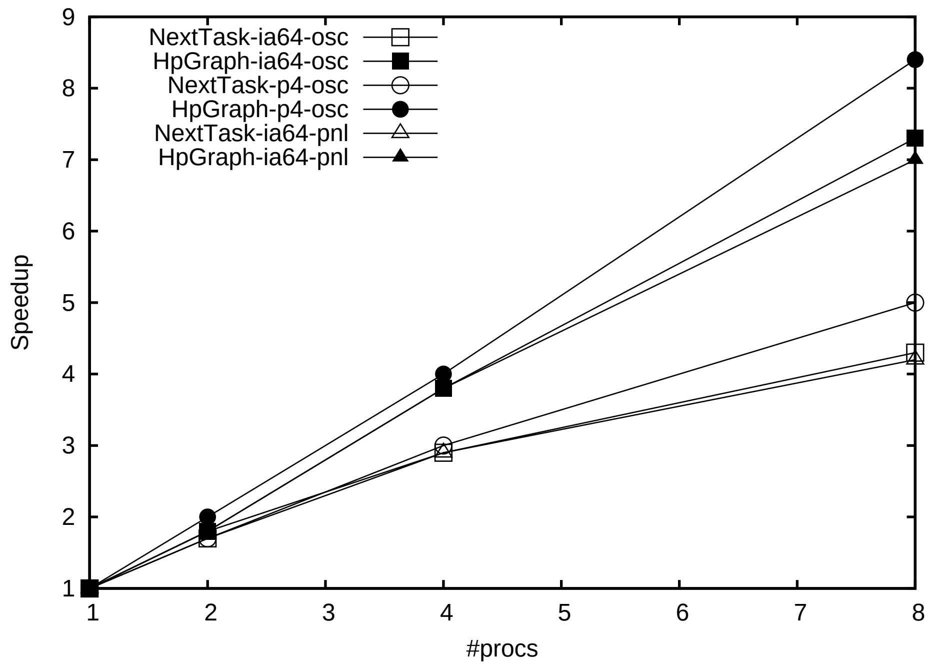


Figure 4.8: Speed-ups for the out-of-core CCD sub-calculation

Figure 4.8 shows the speed-ups, demonstrating the greater scalability of HpGraph, achieving close to linear speed-up. For HpGraph, while the I/O cost decreases with the number of processors, the communication cost increases. Note that NextTask, which uses replicated data, does not have any communication, except while reconciling arrays. The low communication times in p4-osc lead to the observed super-linear speed-up. We intend to investigate communication reduction mechanisms such as overlap of computation and communication to further improve the performance of HpGraph.

The average percentage of total execution time spent performing DGEMM, the core useful computation in the application, is shown in Figure 4.9. It shows the consistent high efficiency achieved by HpGraph, despite the additional overhead of hypergraph partitioning.

System	Scheme	nprocs			
		1	2	4	8
ia64-osc	NextTask	9710	5760	3403	2281
	HpGraph	9244	5110	2408	1271
p4-osc	NextTask	13717	7988	4562	2739
	HpGraph	11700	5886	2899	1390
ia64-pnl	NextTask	7928	4453	2731	1868
	HpGraph	7564	4283	1968	1081

Table 4.1: Turnaround times, in seconds, for the CCD sub-calculation

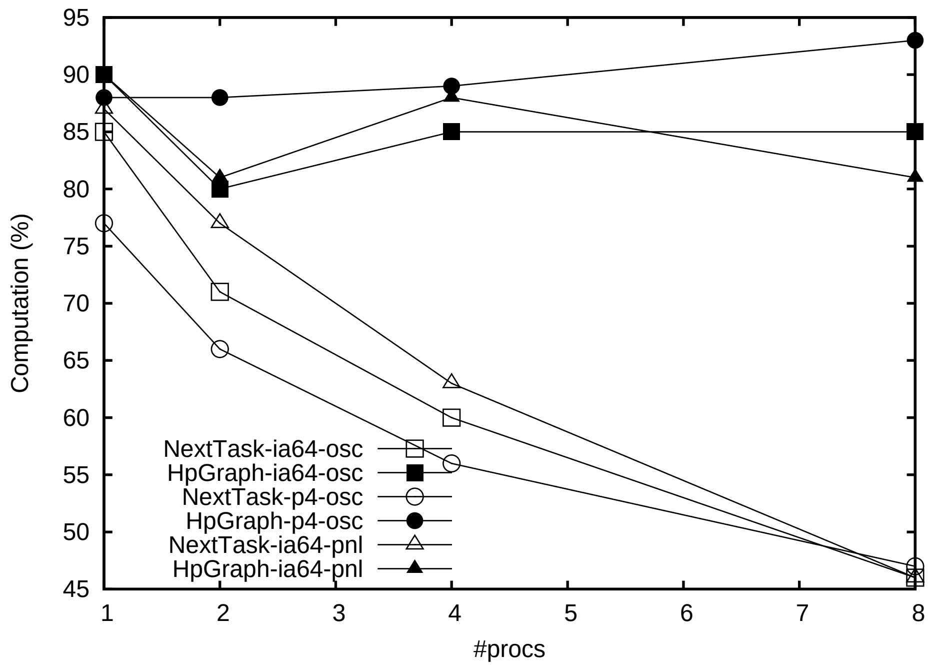


Figure 4.9: Percentage of total time spent in computation for the out-of-core CCD sub-calculation

4.8 Related Work

There is an extensive body of research on compile-time optimization of in-memory computations involving dense matrices, accessed by regular memory reference patterns [4, 63, 78, 79]. These approaches assume dimension-ordered data layout with focus on loop structure determination. There has also been extensive research on optimizing out-of-core computations involving regular data structures such as dense multi-dimensional arrays. Krishnan et al. [65, 66] presented an approach to determining the position of memory allocation and disk I/O in an imperfectly nested loop nest corresponding to fused tensor contraction expressions. Sahoo et al. [96] extended the approach by describing a procedure to efficiently enumerate the various fused loop structures. All such approaches exploit the regularity of the computation as exhibited by the loop structure to optimize disk I/O cost.

Navarro et al. [82] present multi-level blocked algorithms for dense linear algebra computations to minimize TLB misses and page faults incurred by virtual memory. Toledo and Gustavson [110] describe a library that supports blocked data layouts, effective data distribution, and different I/O schemes to implement linear algebra computations on them. Gunter and Van de Geijn [44] present a blocked algorithm for out-of-core QR factorization. These approaches operate on blocked data and exploit the specialized structure of such algorithms. There have also been several out-of-core algorithms derived for other domains [30, 5, 23, 18].

Existing approaches do not readily extend to more general data structures, such as block-sparse arrays. We are not aware of any work that develops integrated compile/runtime approaches for data locality optimization, computational load balancing, and mininization of disk I/O for computations accessing semi-structured and unstructured data.

Brown [17, 81, 16] has proposed mechanisms for compiler-directed memory management of out-of-core applications. Their work points our the limitations of virtual memory and present techniques to automatically manage virtual memory through compiler-directed prefetching. The approach and evaluation is limited to sequential programs. They point to some limitations of using explicit I/O, including the additional burden on the programmer and matching data management to the resources available on the target machine. Our work addresses both these issues by automatically managing disk I/O and taking into account effective disk I/O characteristics and the memory available at runtime.

Abstractions for block-sparse matrices exist in the context of linear algebra and iterative solvers [35, 19]. They provide efficient mechanisms suited for specific computations on block-sparse matrices. We provide a generalized interface for arbitrary computations involving block-sparse matrices.

Our work is similar to Aztec [112], a parallel iterative solver package that provides a global view of a distributed matrix. Advanced partitioning techniques [46] are used to determine the computation distribution and mapping. We provide a general-purpose abstraction for block-sparse matrices. The partitioning of the matrices is performed to balance computation load-balance and communication costs. In addition, the mechanisms provided for locality-aware load-balancing are not tightly coupled with block-sparse matrices, and can be utilized in a wide range of contexts.

There has been extensive research on scheduling task graphs onto processors. One of the earliest work on static scheduling of task graphs is by Sarkar and Hennesy [98]. They model the problem as a clustering problem and put forth compile-time techniques to maximize multi-processor performance by exploiting parallelism while minimizing communication and synchronization overhead. Gerasoulis et al. [40] categorized various clustering algorithms and derived results on linear clustering, upon which Gerasoulis and Yang [41, 122] developed a linear clustering algorithm. McCreary et al. [80] evaluate different clustering algorithms on directed task graphs of certain applications. A survey of static scheduling algorithms was presented by Kwok and Ahmad [68].

Çatalyürek and Aykanat [115, 114] used hypergraph-partitioning to parallelize sparse matrix-vector multiplications. Chang et al. [21] performed parallel data aggregation based on hypergraphs. Khanna et al. [62] present a hypergraph-based approach to scheduling tasks with batch shared I/O.

Parkway [111] is a parallel hypergraph partitioner that interfaces with PaToH [116] and hMETIS [55, 56], which are sequential. To the best of our knowledge, it does not allow a subset of the vertices of the hypergraph to be pre-assigned to some parts. Note that this technique was used to model data distributed in the local memories of processes. We are exploring parallel hypergraph partitioners that support this variation.

Dynamic load-balancing based on work-stealing has been studied, particularly for state-space search [105, 67, 89]. Charm++ [75] supports dynamic load-balancing by object migration. Cilk [12, 94] supports load-balancing of computations based on work-stealing. OpenMP [88] exploits parallelism at the loop level by distributing different iterations to different processors. Locality is not taken into consideration in any of these schemes. The self-scheduling strategy in OpenMP is similar to the *NextTask* scheme that was evaluated earlier.

Our start-time optimizations are similar, in spirit, to the inspector-executor model used in the Chaos tool [97]. Radhakrishnan et al. [104] detailed a dynamic load balancing strategy for applications with slow or abrupt change in their communication and computation patterns. The algorithm incrementally arrives at a better mapping of tasks, allowing refinement of the mapping for iterative computations.

4.9 Conclusion

We designed and implemented high-level abstractions for manipulating block-sparse matrices. Computation primitives to improve load balancing, by exploiting locality, were presented. The programmer exposes the parallelism in the computation, and the system determines the computation mapping. We presented an application of hypergraph partitioning to determining the computation schedule to automatically manage the memory hierarchy. A novel formulation using hypergraph partitioning was presented. Our approach consistently performs better than the alternative schemes considered for load-balancing. Experimental evaluation using a sub-computation from quantum chemistry demonstrates significant improvements in disk I/O cost, overall performance, and scalability.

```
int bi, bj, bk;                                              ▷ Brick sizes. Defined elsewhere
function MATMUL(bsaC, bsaA, bsaB)
    Input: Block-sparse arrays A, B, and C
    TaskPool tpHandle
    LocalityInfo locs[3]
    int nBricksI = bsaGetNBricksAlongDim(bsaC,0);
    int BricksJ = bsaGetNBricksAlongDim(bsaC,1);
    int nBricksK = bsaGetNBricksAlongDim(bsaA,1);
    BsaObject objs[3]={bsaA,bsaB,bsaC}
    AccessMode modes[3]={ACCESS_READ,ACCESS_READ,ACCESS_UPDATE}
    tpHandle = tpCreateTaskPool(objs,modes)                  ▷ Create task pool
    for i=0 to nBricksI - 1 do
        for j=0 to nBricksJ - 1 do
            int cbrick[2] = i,j
            if bsaIsBrickNonZero(bsaC, cbrick) then
                for k=0 to nBricksK - 1 do
                    int abrick[2] = {i,k}
                    int bbrick[2] = {k,j}
                    if bsaIsBrickNonZero(bsaA, abrick) AND bsaIsBrickNonZero(bsaB,
                        bbrick) then
                        int pvt[3]={bi,bj,bk}                ▷ Brick sizes
                        LocalityInfo locs[3] = {abrick, bbrick, cbrick}
                        tpAddTask(tpHandle, BrickMatmul, 3, locs, pvt)
    tpSeal(tpHandle)                                         ▷ Any start-time optimizations
    for i = 0 to maxiter do                                  ▷ Iterative computation
        tpProcess(tpHandle)                                  ▷ Process all tasks, every iteration
    tpDestroy(tpHandle)                                      ▷ Destroy task pool
function BRICKMATMUL(int nLocInfo, LocalityInfo *locs, void *buf[], void *pvt)
    Input: Information on bricks to be multiplied
    int Ni, Nj, Nk, i, j, k
                                     ▷ Actual communication is external to this function.
    double *A = buf[0]                                       ▷ Fetch pointer to data/buffer
    double *B = buf[1]
    double *C = buf[2]
    Ni=pvt[0]; Nj=pvt[1]; Nk=pvt[2];                         ▷ Brick sizes
    for i = 0 to Ni-1 do                                     ▷ Matrix multiplication for this task
        for j = 0 to Nj-1 do
            for k = 0 to Nk-1 do
                C[i,j] += A[i,k]*B[k,j]
```

Algorithm 4.3: Block-sparse matrix multiply. Each task processed by `BrickMatmul`. Task pool is processed `maxiter` times, but is created and sealed once

```
1: for all $p \in P$ do
2:     for all $b \in N_p \cup N_{I_p}$ do
3:         Allocate $b$
4:         if $b \in N_R$ then Read $b$
5:     for all $v \in V_p$ do Compute $v$
6:     for all $b \in N_p \cup N_{I_p}$ do
7:         if $b \in N_W$ then Write $b$
8:         Deallocate $b$
```

Algorithm 4.4: Computation schedule for one-level partitioning

```
1: for all $b \in N_E$ do
2:     Allocate $b$
3:     if $b \in N_R$ then Read $b$
4: for all $p \in P$ do
5:     for all $b \in N_p$ do
6:         Allocate $b$
7:         if $b \in N_R$ then Read $b$
8:     for all $v \in V_p$ do Compute $v$
9:     for all $b \in N_p$ do
10:        if $b \in N_W$ then Write $b$
11:        Deallocate $b$
12: for all $b \in N_E$ do
13:     if $b \in N_W$ then Write $b$
14:     Deallocate $b$
```

Algorithm 4.5: Computation schedule for a read-once partition

```
1: $\Pi \leftarrow$ ReadOnce $(V)$
2: $f \leftarrow$ true
3: for all $p \in P(\Pi)$ do
4:     $f \leftarrow f \wedge (N_p(\Pi) + N_E(\Pi) \leq M)$
5: if $f =$ true then
6:     Return $< V, \Pi >$
7: else
8:     $< V_1, V_2 > \leftarrow$ NetBalancePartition $(V)$
9:     TwoLevel $(V_1) \cup$ TwoLevel $(V_2)$
```

Algorithm 4.6: Two-Level partitioning algorithm: TwoLevel

CHAPTER 5

CONCLUSIONS AND FUTURE DIRECTIONS

Parallel programming is complicated by the numerous details to be handled by the programmer to optimize an application. The details to be handled depend on the application domain and the target system under consideration. In this work we have taken a two-pronged approach to the addressing this problem.

For regular programs consisting of loop nests operating on dense multi-dimensional arrays with constant dependences between the statements, we presented a novel automatic parallelization techniques that require no input from the user. The approach attempts to determine a parallel schedule in which all processes can start execution in parallel, eliminating any start-up overhead. When the concurrent tiled iteration space inhibits such execution we presented techniques to re-enable it in the tiled iteration space.

This approach provides a choice to the programmer in terms of the parallel code generated, enabling optimized parallelization of certain programs that could not be done with existing approaches. We evaluated our approach using the best empirically determined execution parameters for the pipelined schedules. Determining the best execution parameters for overlapped and split tiling could potentially improve its performance further. Our approach currently handles only programs with constant dependences. Extensions to programs with affine dependences will enable handling of a larger class of programs.

For more general computations we presented a programming model with blocked abstractions for data with computation represented as tasks operating on these blocks. When a blocked multi-dimensional array needs to be accessed in a different pattern than the blocking, reblocking it to match the access pattern might be beneficial. We presented algorithms for efficient out-of-core matrix transposition and out-of-core matrix reblocking. The algorithms presented based their I/O sizes on the characteristics of modern storage systems, and minimized total disk I/O volume. A load-balanced schedule was derived for parallel matrix reblocking given a round-robin distribution of data blocks amongst the processors, which is employed by DRA. Extensions to the approach to handle different data distributions, such as one proposed by Toledo and Gustavson [110], could further improve the applicability of out-of-core computation to solve high-end computing problems.

For independent tasks operating on blocked data on disk, we presented abstractions to automatically schedule the computation, communication, and disk I/O so as to ensure load-balanced execution while minimizing data movement overhead. The schedules generated resulted in almost 90% of the time spent in useful computation, rather than data movement, for sub-calculations from Coupled Cluster methods.

Several extensions are possible to the out-of-core abstractions we have presented. Handling tasks with dependences would extend the class of applications that can be handled by the proposed framework. For arbitrary computations operating on blocked data, effective scheduling might not be feasible. We have investigated non-collective parallel I/O on blocked data [64] as an alternative to collective I/O supported by parallel I/O libraries, and achieved results better than replicated processing, even when assuming replication does not incur any cost.

One of the fundamental limitations of out-of-core programming has traditionally been the significant degradation in the bandwidth available, and the performance achieved. For example, Brown [17, 16] shows that the performance of an application degrades significantly once the data set just exceeds the available physical memory size. Our proposed approaches have focused on bridging this gap by ensuring that the secondary storage is effectively used to increase data set sizes supported while incurring a graceful degradation in performance as the problem size transitions from in-memory to out-of-core. This is borne out by our sequential out-of-core matrix reblocking experiments presented in Section 3.4.5 and the out-of-core computation scheduling evaluation in Section 4.7.5. This makes out-of-core programming an attractive alternative to investing in larger supercomputers for scientists primarily interested in very large problem sizes rather than additional processing capacity.

BIBLIOGRAPHY

[1] Vikram Adve, Guohua Jin, John Mellor-Crummey, and Qing Yi. High performance fortran compilation techniques for parallelizing scientific codes. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–23, Washington, DC, USA, 1998. IEEE Computer Society. 2.8

[2] Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Synthesizing transformations for locality enhancement of imperfectly nested loops. In *ICS '00: Proceedings of the 14th international conference on Supercomputing*, pages 141–152, 2000. 2.1

[3] Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Tiling imperfectly-nested loop nests. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, page 31, 2000. 2.1, 2.8

[4] Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. *International Journal of Parallel Programming*, 29(5):493–544, 2001. 2.1, 2.8, 4.8

[5] Deepak Ajwani, Roman Dementiev, and Ulrich Meyer. A computational study of external-memory bfs algorithms. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 601–610, New York, NY, USA, 2006. ACM. 4.8

[6] W. O. Alltop. A computer algorithm for transposing nonsquare arrays. *IEEE Transactions on Computers*, 24(10):1038–1040, 1975. 3.3.1

[7] Corinne Ancourt and François Irigoin. Scanning polyhedra with do loops. In *PPOPP '91: Proceedings of the 3rd ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 39–50, New York, NY, USA, 1991. ACM. 2.6

[8] G. L. Anderson. A stepwise approach to computing the multidimensional fast Fourier transform of large arrays. *IEEE Transactions on Acoustics and Speech Signal Processing*, 28(3):280–284, 1980. 3.1, 3.3.1

[9] Rumen Andonov, Stefan Balev, Sanjay Rajopadhye, and Nicola Yanev. Optimal semi-oblique tiling. *IEEE Transactions on Parallel and Distributed Systems*, 14(9):944–960, 2003. 2.1, 2.8

[10] David H. Bailey. FFTs in external or hierarchical memory. *Journal of Supercomputing*, 4(1):23–35, 1990. 3.1, 3.3.1

[11] Gerald Baumgartner, David E. Bernholdt, Daniel Cociorva, Robert J. Harrison, So Hirata, Chi-Chung Lam, Marcel Nooijen, Russell Pitzer, J. Ramanujam, and P. Sadayappan. A high-level approach to synthesis of high-performance codes for quantum chemistry. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, 2002. 4.2

[12] Michael A. Bender and Michael O. Rabin. Online scheduling of parallel programs on heterogeneous systems with applications to Cilk. *Theory of Computing Systems Special Issue on SPAA '00*, 35:289–304, 2002. 4.8

[13] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, 1995. 3.3.7

[14] Pierre Boulet, Alain Darte, Tanguy Risset, and Yves Robert. (Pen)-ultimate tiling? *Integration, the VLSI Journal*, 17(1):33–51, 1994. 2.1, 2.8

[15] Norman Brenner. Algorithm 467: matrix transposition in place. *ACM Transactions on Mathematical Software*, 16(11):692–694, November 1973. 3.3.2

[16] Angela Demke Brown. *Explicit Compiler-based Memory Management for Out-of-core Applications*. PhD thesis, Carnegie Mellon University, May 2005. 4.8, 5

[17] Angela Demke Brown, Todd C. Mowry, and Orran Krieger. Compiler-based i/o prefetching for out-of-core applications. *ACM Transactions on Computer Systems*, 19(2):111–170, 2001. 3.1, 4.8, 5

[18] Adam L. Buchsbaum, Michael Goldwasser, Suresh Venkatasubramanian, and Jeffery R. Westbrook. On external memory graph traversal. In *SODA '00: Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, pages 859–860, Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics. 4.8

[19] Sandra Carney, Michael A. Heroux, Guangye Li, Roldan Pozo, Karin A. Remington, and Kesheng Wu. A revised proposal for a sparse blas toolkit. Technical Report 94-034, Army High Performance Computing Research Center, June 1994. Updated version at Web address http://www.cray.com/products/applications/support/scal/spblastk.ps. 4.8

[20] Esko G. Cate and David W. Twigg. Algorithm 513: Analysis of in-situ transposition. *ACM Transactions on Mathematical Software*, 3(1):104–110, March 1977. 3.3.2

[21] Chialin Chang, Tahsin Kurc, Alan Sussman, Ümit V. Çatalyürek, and Joel Saltz. A hypergraph-based workload partitioning strategy for parallel data aggregation. In *PPSC '01: Proceedings of the 10th SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, March 2001. 4.8

[22] Ying Chen, Ian T. Foster, Jarek Nieplocha, and Marianne Winslett. Optimizing collective I/O performance on parallel computers: A multisystem study. In *ICS '97: Proceedings of the 11th International Conference on Supercomputing*, 1997. 3.1

[23] Rezaul Alam Chowdhury and Vijaya Ramachandran. External-memory exact and approximate all-pairs shortest-paths in undirected graphs. In *SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 735–744, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics. 4.8

[24] Cristian Coarfa, Yuri Dotsenko, and John Mellor-Crummey. A Multi-Platform Co-Array Fortran Compiler. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques*, 2004. 3.4.1

[25] Daniel Cociorva, Gerald Baumgartner, Chi-Chung Lam, P. Sadayappan, J. Ramanujam, Marcel Nooijen, David E. Bernholdt, , and Robert J. Harrison. Space-time trade-off optimization for a class of electronic structure calculations. In *PLDI '02: Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2002. 4.2

[26] Daniel Cociorva, Xiaoyang Gao, Sandhya Krishnan, Gerald Baumgartner, Chi-Chung Lam, P. Sadayappan, and J. Ramanujam. Global communication optimization for tensor contraction expressions under memory constraints. In *IPDPS '03: Proceedings of the 17th International Parallel & Distributed Processing Symposium*, 2003. 4.2

[27] Daniel Cociorva, John W. Wilkins, Gerald Baumgartner, P. Sadayappan, J. Ramanujam, Marcel Nooijen, David E. Bernholdt, and Robert J. Harrison. Towards automatic synthesis of high-performance codes for electronic structure calculations: Data locality optimization. In *HiPC '01: Proceedings of the 8th Annual International Conference on High Performance Computing*, 2001. 4.2

[28] Daniel Cociorva, John W. Wilkins, Chi-Chung Lam, Gerald Baumgartner, P. Sadayappan, and J. Ramanujam. Loop optimization for a class of memory-constrained computations. In *ICS '01: Proceedings of the 15th international conference on Supercomputing*, pages 500–509, 2001. 4.2

[29] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. In *PLDI '95: Proceedings of the 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 279–290, 1995. 2.1, 2.8

[30] A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. Ramos. Randomized external-memory algorithms for some geometric problems. In *SCG '98: Proceedings of the fourteenth annual symposium on Computational geometry*, pages 259–268, New York, NY, USA, 1998. ACM. 4.8

[31] T. Crawford and H. Schaefer III. An Introduction to Coupled Cluster Theory for Computational Chemists. In K. Lipkowitz and D. Boyd, editor, *Reviews in Computational Chemistry*, volume 14, pages 33–136. John Wiley & Sons, Ltd., 2000. 4.1, 4.2, 4.7.5

[32] Roman Dementiev and Peter Sanders. Asynchronous parallel disk sorting. In *SPAA '03: Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 138–148, New York, NY, USA, 2003. ACM. 3.5

[33] Frederic Desprez, Jack Dongarra, Fabrice Rastello, and Yves Robert. Determining the idle time of a tiling: new results. *Journal of Information Science and Engineering*, 14:167–190, 1998. 2.1, 2.8

[34] Jack Dongarra and Robert Schreiber. Automatic blocking of nested loops. Technical report, University of Tennessee, Knoxville, TN, August 1990. 2.1, 2.8

[35] Iain S. Duff, Michele Marrone, Giuseppe Radicati, and Carlo Vittoli. Level 3 basic linear algebra subprograms for sparse matrices: a user-level interface. *ACM Transactions on Mathematical Software*, 23(3):379–401, 1997. 4.8

[36] Alan Edelman, Steve Heller, and S. Lennart Johnsson. Index transformation algorithms in a linear algebra framework. *IEEE Transactions on Parallel and Distributed Systems*, 5(12):1302–1309, 1994. 3.3.3

[37] J. O. Eklundh. A fast computer method for matrix transposing. *IEEE Transactions on Computers*, 20(7):801–803, 1972. 3.3.1, 3.3.2, 3.3.4, 3.5

[38] Ian Foster and Jarek Nieplocha. Disk Resident Arrays: An array-oriented I/O library for out-of-core computations. In Rajkumar Buyya, Hai Jin, and Toni Cortes, editors, *Disk Arrays and Parallel I/O: Theory and Practice*. IEEE Computer Society Press, 2001. 3.1

[39] Matteo Frigo and Volker Strumpen. The memory behavior of cache oblivious stencil computations. *Journal of Supercomputing*, 39(2):93–112, 2007. 2.1, 2.8

[40] Apostolos Gerasoulis, Sesh Venugopal, and Tao Yang. Clustering task graphs for message passing architectures. *SIGARCH Computer Architecture News*, 18(3):447–456, 1990. 4.8

[41] Apostolos Gerasoulis and Tao Yang. On the granularity and clustering of directed acyclic task graphs. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):686–701, 1993. 4.8

[42] Martin Griebl. On tiling space-time mapped loop nests. In *SPAA '01: Proceedings of the 13th annual ACM symposium on Parallel algorithms and architectures*, pages 322–323, 2001. 2.1, 2.8

[43] Martin Griebl. *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. University of Passau, 2004. Habilitation Thesis. 2.1, 2.8

[44] Brian C. Gunter and Robert A. Van De Geijn. Parallel out-of-core computation and updating of the qr factorization. *ACM Transactions on Mathematical Software*, 31(1):60–78, 2005. 4.8

[45] R.M. Haralick and L.G. Shapiro. *Computer and Robot Vision*. Addison Wesley, 1992. 2.1

[46] Bruce Hendrickson and Robert Leland. The chaco user's guide: Version 2.0. Technical Report SAND94–2692, Sandia National Laboratories, 1994. 4.8

[47] High Performance Computational Chemistry Group. *"NWChem, A Computational Chemistry Package for Parallel Computers, Version 4.6"*. Pacific Northwest National Laboratory, Richland, Washington 99352–0999, USA, 2004. 3.4.1, 4.7.5

[48] So Hitara. Tensor contraction engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories. *The Journal of Physical Chemistry*, 107(46):9887–9897, 2003. 4.6.3

[49] Edin Hodzic and Weijia Shang. On time optimal supernode shape. *IEEE Transactions on Parallel and Distributed Systems*, 13(12):1220–1233, 2002. 2.1, 2.8

[50] Karin Högstedt, Larry Carter, and Jeanne Ferrante. Determining the idle time of a tiling. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 160–173, 1997. 2.1, 2.8

[51] Karin Högstedt, Larry Carter, and Jeanne Ferrante. Selecting tile shape for minimal execution time. In *SPAA '99: Proceedings of the 11th annual ACM symposium on Parallel algorithms and architectures*, pages 201–211, New York, NY, USA, 1999. ACM. 2.1, 2.8

[52] François Irigoin and Rémi Triolet. Supernode partitioning. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 319–329, New York, NY, USA, 1988. ACM. 2.1, 2.2, 2.8

[53] Shoaib Kamil, Kaushik Datta, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Implicit and explicit optimizations for stencil computations. In *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 51–60, New York, NY, USA, 2006. ACM. 2.1, 2.8

[54] Shoaib Kamil, Parry Husbands, Leonid Oliker, John Shalf, and Katherine Yelick. Impact of modern memory subsystems on cache optimizations for stencil computations. In *MSP '05: Proceedings of the 2005 workshop on Memory system performance*, pages 36–43, New York, NY, USA, 2005. ACM. 2.1, 2.8

[55] George Karypis, Rajat Aggrawal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: Applications in VLSI domain. In *DAC '97: Proceedings of 34th ACM/IEEE Design Automation Conference*, 1997. 4.5, 4.8

[56] George Karypis and Vipin Kumar. Multilevel *k*-way hypergraph partitioning. In *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 343–348, New York, NY, USA, 1999. ACM. 4.8

[57] S. D. Kaushik, Chua-Huang Huang, John R. Johnson, Rodney W. Johnson, and P. Sadayappan. Efficient transposition algorithms for large matrices. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 656–665. ACM Press, 1993. 3.3.1, 3.3.2, 3.5

[58] Ramakrishnan Kazhiyur-Mannar, Rephael Wenger, Roger Crawfis, and Tamal K. Dey. Adaptive resolution isosurface construction in three and four dimensions. Technical Report OSU-CISRC-7/03–TR38, Department of Computer and Information Science, The Ohio State University, July 2003. 3.1

[59] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. The omega library interface guide. Technical report, Univ. of Maryland Institute for Advanced Computer Studies Report No. UMIACS-TR-95-41, College Park, MD, USA, 1995. 2.8

[60] Wayne Kelly, William Pugh, and Evan Rosser. Code generation for multiple mappings. In *FRONTIERS '95: Proceedings of the 5th Symposium on the Frontiers of Massively Parallel Computation*, page 332, Washington, DC, USA, 1995. IEEE Computer Society. 2.1

[61] Wayne Kelly, William Pugh, Evan Rosser, and Tatiana Shpeisman. Transitive closure of infinite graphs and its applications. *International Journal of Parallel Programming*, 24(6):579–598, 1996. 2.8

[62] Gaurav Khanna, Nagavijayalakshmi Vydyanathan, Tahsin Kurc, Ümit V. Çatalyürek, Pete Wyckoff, Joel Saltz, and P. Sadayappan. A Hypergraph Partitioning Based Approach for Scheduling of Tasks with Batch-shared I/O. In *CCGrid '05: Proceedings of the 5th IEEE/ACM International Symposium on Cluster Computing and the Grid*, May 2005. 4.8

[63] Indraprakas Kodukula, Nawaz Ahmed, and Keshav Pingali. Data-centric multi-level blocking. In *PLDI '97: Proceedings of the 1992 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 346–357, 1997. 4.8

[64] Sriram Krishnamoorthy, Juan Piernas Canovas, Vinod Tipparaju, Jarek Nieplocha, and P. Sadayappan. Non-collective parallel i/o for global address space programming models. In *CLUSTER '07: Proceedings of the International Conference on Cluster Computing*. IEEE Computer Society Press, December 2007. 5

[65] Sandhya Krishnan, Sriram Krishnamoorthy, Gerald Baumgartner, Daniel Cociorva, Chi-Chung Lam, P. Sadayappan, J. Ramanujam, David E. Bernholdt, and Venkatesh Choppella. Data locality optimization for synthesis of efficient out-of-core algorithms. In *HiPC '03: Proceedings of the 10th Annual International Conference on High Performance Computing*, pages 406–417. Spring Verlag, December 2003. 4.8

[66] Sandhya Krishnan, Sriram Krishnamoorthy, Gerald Baumgartner, Chi-Chung Lam, J. Ramanujam, P. Sadayappan, and Venkatesh Choppella. Efficient synthesis of out-of-core algorithms using a nonlinear optimization solver. *Journal on Parallel and Distributed Computing*, 66(5):659–673, 2006. 4.8

[67] Vipin Kumar, K. Ramesh, and V. Nageshwara Rao. Parallel best-first search of state-space graphs: a summary of results. In *National Conference on Artificial Intelligence*, pages 122–127, 1988. 4.8

[68] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, 1999. 4.8

[69] Susan Laflin and M. A. Brebner. Algorithm 380: in-situ transposition of a rectangular matrix. *Communications of the ACM*, 13(5):324–326, May 1970. 3.3.2

[70] Chi-Chung Lam. *Performance Optimization of a Class of Loops Implementing Multi-Dimensional Integrals*. PhD thesis, The Ohio State University, Columbus, OH, August 1999. 4.2

[71] Chi-Chung Lam, Daniel Cociorva, Gerald Baumgartner, and P. Sadayappan. Memory-optimal evaluation of expression trees involving large objects. In *HiPC '99: Proceedings of the 6th International Conference on High Performance Computing*, pages 103–110, London, UK, 1999. Springer-Verlag. 4.2

[72] Chi-Chung Lam, Daniel Cociorva, Gerald Baumgartner, and P. Sadayappan. Optimization of Memory Usage and Communication Requirements for a Class of Loops Implementing Multi-Dimensional Integrals. In *LCPC '99: Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, pages 350–364. Springer Verlag, 1999. 4.2

[73] Chi-Chung Lam, P. Sadayappan, and Rephael Wenger. On Optimizing a Class of Multi-Dimensional Loops with Reductions for Parallel Execution. *Parallel Processing Letters*, 7(2):157–168, 1997. 4.2

[74] Chi-Chung Lam, P. Sadayappan, and Rephael Wenger. Optimization of a Class of Multi-Dimensional Integrals on Parallel Machines. In *PPSC '97: Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997. 4.2

[75] Laxmikant V. Kalé and Sanjeev Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *OPOSLA '93: Proceedings of the 8th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 91–108. ACM Press, September 1993. 4.8

[76] Thomas Lengauer. *Combinatorial algorithms for integrated circuit layout*. John Wiley & Sons, Inc., New York, NY, USA, 1990. 4.5

[77] Zhiyuan Li and Yonghong Song. Automatic tiling of iterative stencil loops. *ACM Transactions on Programming Languages and Systems*, 26(6):975–1028, 2004. 2.8

[78] Amy W. Lim and Monica S. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing*, 24(3-4):445–475, 1998. 4.8

[79] Amy W. Lim, Shih-Wei Liao, and Monica S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *PPoPP '01: Proceedings of the 8th ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 103–112, New York, NY, USA, 2001. ACM. 4.8

[80] Carolyn McCreary, A. A. Khan, J. J. Thompson, and M. E. McArdle. A comparison of heuristics for scheduling dags on multiprocessors. In *IPPS '94: Proceedings of the 8th International Parallel Processing Symposium*, pages 446–451, Washington, DC, USA, 1994. IEEE Computer Society. 4.8

[81] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic compiler-inserted i/o prefetching for out-of-core applications. In *OSDI '96: Proceedings of the second USENIX symposium on Operating systems design and implementation*, pages 3–17, New York, NY, USA, 1996. ACM. 4.8

[82] Juan J. Navarro, Toni Juan, and Tomas Lang. MOB Forms: A Class of Multilevel Block Algorithms for Dense Linear Algebra Operations. In *ICS '94: Proceedings of the 8th International Conference on Supercomputing*, 1994. 4.8

[83] Jarek Nieplocha and Bryan Carpenter. ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems. In *RTSPP '99: Proceedings of the 3rd Workshop on Runtime Systems for Parallel Programming*, 1999. 3.4.1

[84] Jarek Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global arrays: a portable programming model for distributed memory computers. In *Supercomputing '94: Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, pages 340–349, 1994. 3.4.1

[85] Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease, and Edo Apra. Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit. *International Journal High Performance Computing and Applications*, to appear, 2005. 3.4.1

[86] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global Arrays: A nonuniform memory access programming model for high-performance computers. *Journal of Supercomputing*, 10(2):169–189, 1996. 3.4.1

[87] Ohio Supercomputing Center. http://www.osc.edu. 3.2

[88] OpenMP Specification. http://www.openmp.org/specs. 4.6.3, 4.8

[89] R. Panwar, W. Kim, and Gul Agha. Parallel implementations of irregular problems using high-level actor language. In *IPPS '96: Proceedings of the 10th International Parallel Processing Symposium*, pages 857–862, Washington, DC, USA, 1996. IEEE Computer Society. 4.8

[90] K. Parzyszek, Jarek Nieplocha, and R. A. Kendall. A Generalized Portable SHMEM Library for High Performance Computing. In *Proceedings of the IASTED Parallel and Distributed Computing and Systems*, pages 401–406, November 2000. 3.4.1

[91] William Pugh and David Wonnacott. Static analysis of upper and lower bounds on dependences and parallelism. *ACM Transactions on Programming Languages and Systems*, 16(4):1248–1278, 1994. 2.8

[92] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for nonshared memory machines. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 111–120, 1991. 2.1, 2.8

[93] H. K. Ramapriyan. A generalization of Eklundh's algorithm for transposing large matrices. *IEEE Transactions on Computers*, 24(12):1221–1226, 1975. 3.3.1

[94] Keith H. Randall. *Cilk: Efficient Multithreaded Computing*. PhD thesis, MIT Department of Electrical Engineering and Computer Science, June 1998. 4.8

[95] Lakshminarayanan Renganarayana and Sanjay Rajopadhye. A geometric programming framework for optimal multi-level tiling. In *Supercomputing '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 18, Washington, DC, USA, 2004. IEEE Computer Society. 2.1, 2.8

[96] Swarup Kumar Sahoo, Sriram Krishnamoorthy, Rajkiran Panuganti, and P. Sadayappan. Integrated loop optimizations for data locality enhancement of tensor contraction expressions. In *Supercomputing '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, 2005. 4.8

[97] Joel Saltz, Ravi Ponnusamy, Shamik Sharma, Bongki Moon, and Raja Das. A manual for the CHAOS runtime library. Technical Report CS-TR-3437 and UMIACS-TR-95-34, University of Maryland, Department of Computer Science and UMIACS, March 1995. 4.8

[98] Vivek Sarkar and John Hennessy. Compile-time partitioning and scheduling of parallel programs. In *SIGPLAN '86: Proceedings of the 1986 SIGPLAN symposium on Compiler construction*, pages 17–26, New York, NY, USA, 1986. ACM. 4.8

[99] Aaron Sawdey and Matthew T. O'Keefe. Program analysis of overlap area usage in self-similar parallel programs. In *LCPC '97: Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computing*, pages 79–93, 1998. 2.1, 2.8

[100] Aaron Sawdey, Matthew T. O'Keefe, and Rainer Bleck. The design, implementation, and performance of a parallel ocean circulation model. In *Proceedings of 6th ECMWF Workshop on the Use of Parallel Processors in Meteorology: Coming of Age*, pages 523–550, 1995. 2.1

[101] Steven W. Schlosser, Jiri Schindler, Stratos Papadomanolakis, Minglong Shao, Anastassia Ailamaki, Christos Faloutsos, and Gregory R. Ganger. On multidimensional data and modern disks. In *FAST '05: Proceedings of the 4th USENIX Conference on File and Storage Technology*, 2005. 3.5

[102] Kent E. Seamons and Marianne Winslett. Multidimensional array I/O in Panda 1.0. *Journal of Supercomputing*, 10(2):191–211, 1996. 3.1

[103] Minglong Shao, Steven W. Schlosser, Stratos Papadomanolakis, Jiri Schindler, Anastassia Ailamaki, Christos Faloutsos, and Gregory R. Ganger. Multimap: Preserving disk locality for multidimensional datasets. In *ICDE '07: Proceedings of the IEEE 23rd International Conference on Data Engineering*, April 2007. 3.5

[104] Robert K. Brunner Shobana Radhakrishnan and Laxmikant V. Kalé. Branch and bound based load balancing for parallel applications. Technical Report 99-06, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, May 1999. 4.8

[105] Amitabh B. Sinha and Laxmikant V. Kalé. A load balancing strategy for prioritized execution of tasks. In *IPPS '93: Proceedings of the 7th International Parallel Processing Symposium*, pages 230–237, Newport Beach, CA., April 1993. 4.8

[106] Yonghong Song and Zhiyuan Li. New tiling techniques to improve cache temporal locality. In *PLDI '99: Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 215–228, 1999. 2.1, 2.8

[107] Jinwoo Suh and Viktor K. Prasanna. An efficient algorithm for out-of-core matrix transposition. *IEEE Transactions on Computers*, 51(4):420–438, April 2002. 3.3.1

[108] Allen Taflove and Susan C. Hagness. *Computational Electrodynamics: The Finite-Difference Time-Domain Method, Third Edition*. Artech House Publishers, 2005. 2.1

[109] The Panda Project – Data Management for High-Performance Scientific Computation. http://drl.cs.uiuc.edu/panda/. 3.1

[110] Sivan Toledo and Fred G. Gustavson. The design and implementation of solar, a portable library for scalable out-of-core linear algebra computations. In *IOPADS '96: Proceedings of the fourth workshop on I/O in parallel and distributed systems*, pages 28–40, New York, NY, USA, 1996. ACM. 4.8, 5

[111] Aleksandar Trifunovic and William J. Knottenbelt. Parkway2.0: A parallel multilevel hypergraph partitioning tool. In *Proceedings of the 19th International Symposium on Computer and Information Sciences*, volume 3280 of *Lecture Notes in Computer Science*, pages 789–800. Spring Verlag, 2004. 4.8

[112] Ray S. Tuminaro, Mike Heroux, Scott A. Hutchinson, and John N. Shadid. Official aztec user's guide: Version 2.1. Technical report, Sandia National Laboratories, 1999. 4.8

[113] R. E. Twogood and M. P. Ekstrom. An extension of Eklundh's matrix transposition algorithm and its application to digital signal processing. *IEEE Transactions on Computers*, 25(12):950–952, 1976. 3.3.1

[114] Ümit V. Çatalyürek and Cevdat Aykanat. Decomposing irregularly sparse matrices for parallel matrix-vector multiplications. In *Irregular '96: Proceedings of 3rd International Symposium on Solving Irregularly Structured Problems in Parallel*, volume 1117 of *Lecture Notes in Computer Science*, pages 75–86. Spring Verlag, 1996. 4.8

[115] Ümit V. Çatalyürek and Cevdat Aykanat. Hypergraph-partitioning based decomposition for parallel spars e-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, 1999. 4.5, 4.8

[116] Ümit V. Çatalyürek and Cevdat Aykanat. *PaToH: A Multilevel Hypergraph Partitioning Tool, Version 3.0.* Bilkent University, Department of Computer Engineering, Ankara, 06533 Turkey. PaToH is available at http://bmi.osu.edu/∼umit/software.htm, 1999. 4.8

[117] Jeffrey Scott Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001. 3.5

[118] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *PLDI '91: Proceedings of the 1991 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 30–44, 1991. 2.1, 2.8

[119] Michael Wolfe. More iteration space tiling. In *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 655–664, New York, NY, USA, 1989. ACM. 2.1, 2.8

[120] David Wonnacott. Time skewing for parallel computers. In *LCPC '99: Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, pages 477–480, London, UK, 2000. Springer-Verlag. 2.8

[121] David Wonnacott. Achieving scalable locality with time skewing. *International Journal of Parallel Programming*, 30(3):181–221, 2002. 2.8

[122] Tao Yang and Apostolos Gerasoulis. Pyrros: static task scheduling and code generation for message passing multiprocessors. In *ICS '92: Proceedings of the 6th International Conference on Supercomputing*, pages 428–437, New York, NY, USA, 1992. ACM Press. 4.8