Hypergraph Partitioning for Automatic Memory Hierarchy Management

Sriram Krishnamoorthy[1] Umit Catalyurek[2] Jarek Nieplocha[3]
Atanas Rountev[1] P. Sadayappan[1]
[1] Dept. of Computer Science and Engineering, [2] Dept. of Biomedical Informatics
The Ohio State University
[3] Pacific Northwest National Laboratory

Abstract

In this paper, we present a mechanism for automatic management of the memory hierarchy, including secondary storage, in the context of a global address space parallel programming framework. The programmer specifies the parallelism and locality in the computation. The scheduling of the computation into stages, together with the movement of the associated data between secondary storage and global memory, and between global memory and local memory, is automatically managed. A novel formulation of hypergraph partitioning is used to model the optimization problem of minimizing disk I/O. Experimental evaluation of the proposed approach using a sub-computation from the quantum chemistry domain shows a reduction in the disk I/O cost by up to a factor of 11, and a reduction in turnaround time by up to 49%, as compared to alternative approaches used in state-of-the-art quantum chemistry codes.

1 Introduction

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 amounts 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 (such as memory hierarchy, processor association, etc.) The higher level abstractions improve programmer productivity.

While existing global address space models simplify the task of programming parallel applications, the problem is compounded when programming out-of-core applications. In addition to handling the various aspects of parallelism, a programmer has to contend with the choice of placement of data in the memory hierarchy, 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.

In this paper, we present an approach to automatic management of data movement and scheduling of computation for *out-of-core* programs, those with data too large to fit into the collective physical memory of the parallel system. This work is done in the context of a global address space framework for programming parallel out-of-core applications. To improve productivity, the framework presents the user with a computation abstraction that allows her to express the 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. In particular, in this paper, we demonstrate the approach to automatic memory hierarchy management using block-sparse arrays that arise in quantum chemistry calculations such as Coupled Cluster methods [Crawford and III 2000].

There has been extensive research to optimize out-of-core computations involving regular data structures such as dense multi-dimensional arrays [Navarro et al. 1994; Sahoo et al. 2005; Krishnan et al. 2003; Krishnan et al. 2004]. A typical solution in the case of dense multi-dimensional arrays is tiling. Existing approaches do not readily extend to more general data structures, such as block-sparse arrays.

We use hypergraph partitioning for scheduling 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. The performance advantages over alternative approaches employed in the state-of-the-art implementation of computation chemistry models in NWChem [High Performance Computational Chemistry Group 2004] are demonstrated.

The paper is organized as follows. Section 2 describes the computational context. Section 3 distinguishes the work in this paper from other related work. The data and computation abstractions that form the basis of this work are briefly described in Section 4. The hypergraph partitioning scheme is detailed in Section 5. Experimental evaluation of our scheme is presented in Section 6. Section 7 concludes the paper.

2 Motivation

One of the major motivations for the development of the proposed approach is the quantum chemistry models such as Coupled Cluster methods [Crawford and III 2000]. Tensor Contraction Engine (TCE) [Baumgartner et al. 2002] 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. 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:

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

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 so as 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 total disk I/O cost is minimized.

3 Related Work

Abstractions for block-sparse matrices exist in the context of linear algebra and iterative solvers [Duff et al. 1997]. Aztec [Tuminaro et al. 1999] is a parallel iterative solver package that provides a global view of a distributed matrix. Advanced partitioning techniques [Hendrickson and Leland 1994] are used to determine the computation distribution and mapping. Our goal is to provide a general-purpose abstraction for implementing parallel algorithms operating on semi-structured data, with block-sparse matrices as the first data structure we have targeted. The computation abstractions and the mechanisms for optimization are not tightly coupled with block-sparse matrices, and can be utilized in a wide range of contexts.

Dynamic load-balancing based on work-stealing has been studied, particularly for state-space search [Sinha and Kalé 1993] . Charm++ [Kalé and Krishnan 1993] supports dynamic load-balancing by object migration. Cilk [Randall 1998] supports load-balancing of computations based on work-stealing. OpenMP exploits parallelism at the loop level by distributing different iterations to different processors. Data locality is not explicitly taken into consideration in any of these systems.

Çatalyürek and Aykanat [Çatalyürek and Aykanat 1999; Çatalyürek and Aykanat 1996] have used hypergraph-partitioning to parallelize sparse matrix-vector multiplications. Chang et al. [Chang et al. 2001] performed parallel data aggregation based on hypergraphs. Khanna et al. [Khanna et al. 2005] present a hypergraph-based approach to scheduling tasks with batch shared I/O. All these and many other applications of hypergraph partitioning focus on obtaining an effective mapping of a parallel computation on to the processors of a parallel system. We focus on using hypergraph partitioning for scheduling, i.e., for sequencing of computations and data movement - not a natural match to hypergraph partitioning.

Our start-time optimizations are similar, in spirit, to the inspector-executor model used in Chaos [Saltz et al. 1995]. There is an extensive body of research on optimizing computations involving dense matrices, accessed by regular memory reference patterns [Ahmed et al. 2000; Kodukula et al. 1997; Lim and Lam 1998; Lim et al. 2001; Krishnan et al. 2003; Krishnan et al. 2004]. We are not aware of any work that develops integrated compile/runtime approaches for data locality optimization, computational load balancing, and minimization of disk I/O for computations accessing semi-structured and unstructured data.

4 Locality-Aware Abstractions

In this section, we briefly discuss the data and computation abstractions that constitute the global address space framework under consideration. Note that the computation abstraction is decoupled from the data abstractions and can be leveraged for other data structures as well. The details of these abstractions can be found in [Krishnamoorthy et al. 2006].

4.1 Abstraction for Block-Sparse Matrices

An abstraction is provided to manipulate multi-dimensional block-sparse matrices that occur in the context of the TCE. The user specifies a brick size along each dimension, which is used to divide the dimensions of the array, so that the non-zero blocks can be stored as collections of bricks. Each brick is uniquely identified by a brick number, derived from its position in the array. A brick is the basic unit of communication and I/O. A disk array is provided that distributes the bricks amongst the local disks of the processors. Collective I/O operations enable reading from and writing to collections of bricks in the disk arrays.

The disk array has an in-memory counterpart — the memory brick collection. The memory brick collection is a global distributed data structure that supports one-sided communication to an arbitrary brick in the collection, given the brick number.

Manipulating a data array involves creating a memory brick collection and populating it with the set of bricks to be accessed by that memory brick collection. Memory is then allocated to accommodate the bricks. Collective I/O operations can now be used to move all the bricks in the memory brick collection between the global memory and disk.

4.2 Abstraction For Locality-Aware Load Balancing

The computation abstraction provided to the user enables the specification of a set of independent tasks, without any dependences, to be executed in parallel. Each task is sequential and is associated with a set of data elements in a globally addressed data structure such as the one described above. An estimate of the execution time of each task is also specified. The task is processed using a user-supplied function that is assumed to be optimized for sequential execution.

A majority of practical parallel applications have outer serializing loops (representing sequencing in time or iteration till convergence), but within those outer-serial loops, they exhibit considerable “forall” parallelism. A significant number of engineering codes using finite-element, finite-difference, and finite-volume methods fit this model. For these applications, an abstraction of independent tasks within each iteration of the outer loop is appropriate, as is the case for the TCE application that motivates this work.

A *task pool* is created and populated with all the tasks to be processed. Before starting the processing of any task in the task pool, the task pool is sealed to signify the completion of population operations. At this stage, the tasks in the task pool are analyzed for data reuse and a schedule for I/O and computation is determined.

The computation and I/O schedule, once determined, can be used to process the set of tasks in the task pool multiple times. For example, in TCE, a given sequence of tensor contractions is evaluated many times for convergence. Thus the start-time cost of optimization is paid once and is amortized over multiple executions of the task pool.

All global data structures are initially assumed to be distributed amongst the local disks attached to the processors in a cluster. Movement of data from the distributed data structures in disk to their in-memory counterparts is done collectively. Once the data is in the global memory, the computation proceeds asynchronously with each process evaluating the next task in the sequence of tasks to be executed.

The memory left unused after allocating the distributed data structures is used to accommodate a LRU cache. The cache reduces the overall communication cost and network contention in the system.

5 Hypergraph Partitioning

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 explain the hypergraph partitioning problem. The limitations of a direct application of the hypergraph partitioning model are discussed. We then present an alternative formulation that better solves our problem of interest.

5.1 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 data elements (henceforth called bricks) need to be allocated and de-allocated, whereas the 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 aggregate memory available is the constraint imposed on the I/O schedule.

5.2 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 by a set of vertices V and a set of nets (hyper-edges) N among those vertices, where 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. The two relevant ones for our context are cut-net and connectivity-1, defined as follows:

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

$$\chi(\Pi) = \sum_{n_j \in N_E} c_j(\lambda_j - 1) \tag{2}$$

With the cut-net metric (1), each cut net n_j contributes its cost to the cut, whereas with the connectivity-1 metric (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 multilevel paradigm, such as hMETIS [Karypis et al. 1997] and PaToH [Çatalyürek and Aykanat 1999], have been shown to compute good partitions quickly for this NP-hard problem.

5.3 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 communication cost incurred by 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.

Fig. 1 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

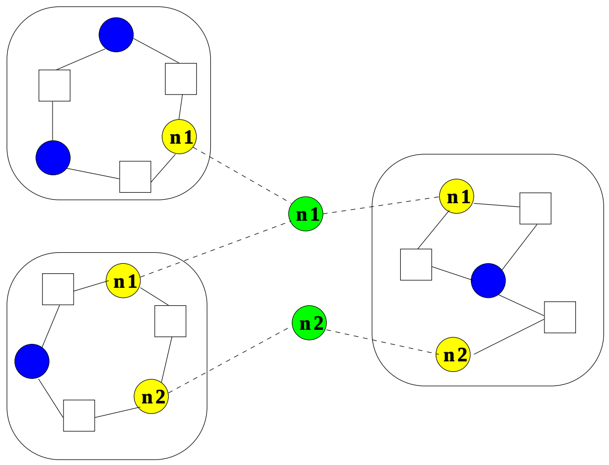

Figure 1: Illustration of one-level partitioning

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 1. 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.

Algorithm 1 Computation schedule for One-Level Partitioning

```
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 $b$
```

5.4 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 2.

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., $cutsize + C_p \leq memory\_limit$ 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.

Algorithm 2 Computation schedule for a Read-Once Partition

```
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 $b$
```

5.5 Integrated Approach: Two-Level Partitioning

The integrated algorithm, **TwoLevel**, is shown in Algorithm 3. 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 routine denoted by **NetBalancePartition**, 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.

Fig. 2 shows a possible partitioning of the same computation as in Fig. 1 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 Fig. 1. Two of the stages produced by the 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 Fig. 1.

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.

Algorithm 3 Two-Level Partitioning Algorithm: TwoLevel

```
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 $(V_2)$
```

Definition 1. *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

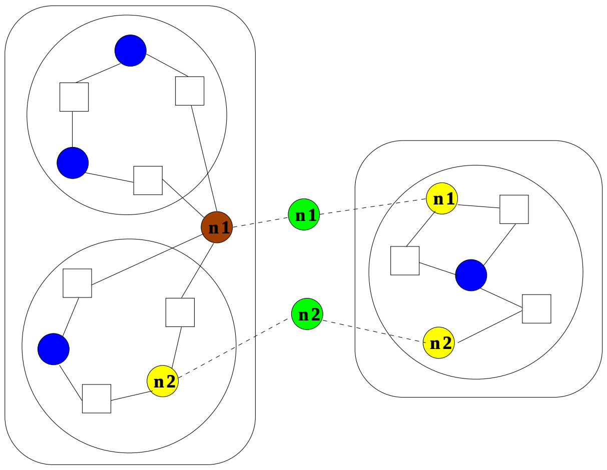

Figure 2: Illustration of two-level partitioning

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 that the two-level approach. □

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. □

In the experimental evaluation, we will focus on evaluating the two-level partitioning scheme.

6 Experimental Evaluation

We evaluate our approach in the quantum chemistry domain described in Section 2. In particular, we use the following Coupled Cluster Doubles (CCD) [Crawford and III 2000] sub-computation:

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

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 [High Performance Computational Chemistry Group 2004]. 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 will be referred to as *GetNext*, in the spirit of the computation distribution scheme adopted in it. The inputs are assumed to be replicated when evaluating this scheme. A reconcile operation is carried out on $i1$ before it is consumed to produce $i0$. In addition, the output array $i0$ is reconciled as the final step. All inputs are assumed to be distributed when 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 unutilized 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.

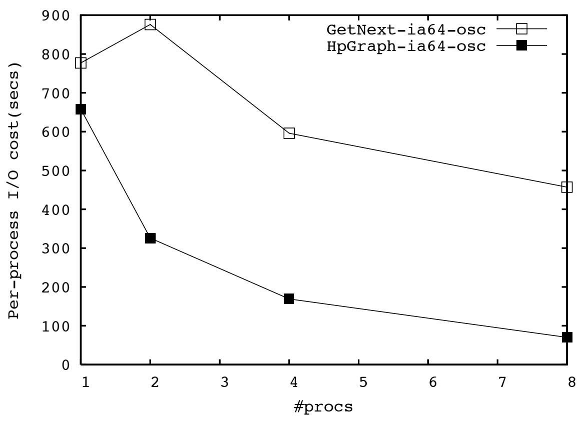

Figure 3: Average per-process I/O cost, in seconds, on ia64-osc

The sub-computation 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. 3, 4, and 5, 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 GetNext on ia64-osc is due to an ineffective task distribution that 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 GetNext 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 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 GetNext. On p4-osc for eight processors, HpGraph leads to a 49% improvement over GetNext, with similar trends observed for other processes. Note that the input arrays are assumed to be replicated for the GetNext scheme. The

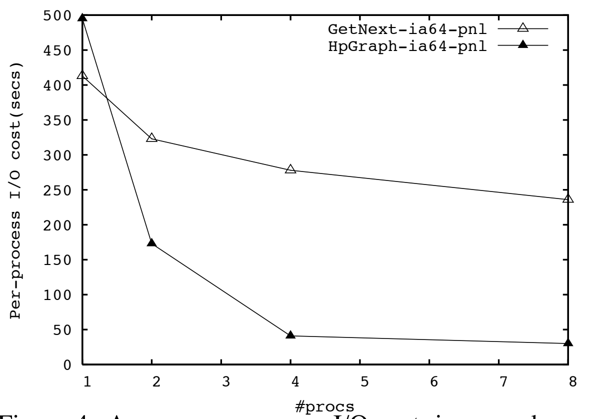

Figure 4: Average per-process I/O cost, in seconds, on ia64-pnl

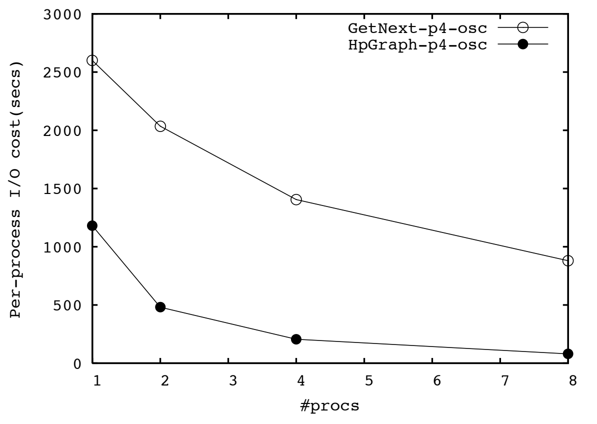

Figure 5: Average per-process I/O cost, in seconds, on p4-osc

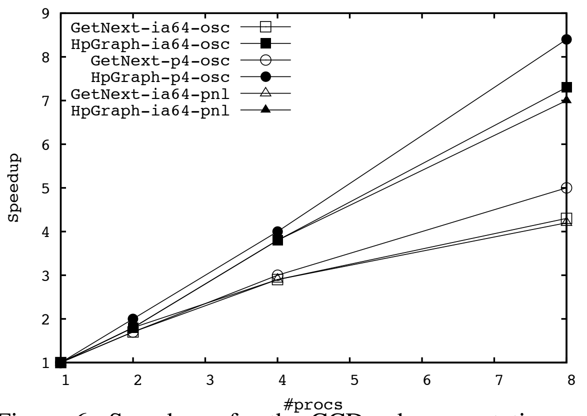

Figure 6: Speed-ups for the CCD sub-computation on the three systems.

Table 1: Turnaround times, in seconds, for the CCD sub-computation

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

improvements obtained would be even higher if the cost of replicating the input arrays is included in the execution time of GetNext.

The speed-ups are shown in Fig. 6. The HpGraph scheme achieves close to linear speed-up, a significant improvement over GetNext. For HpGraph, while the I/O cost decreases with the number of processors, the communication cost increases. Note that GetNext, which uses replicated data, does not incur any communication costs, 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 Fig. 7. It shows the consistent high efficiency achieved by HpGraph, despite the additional overhead of hypergraph partitioning.

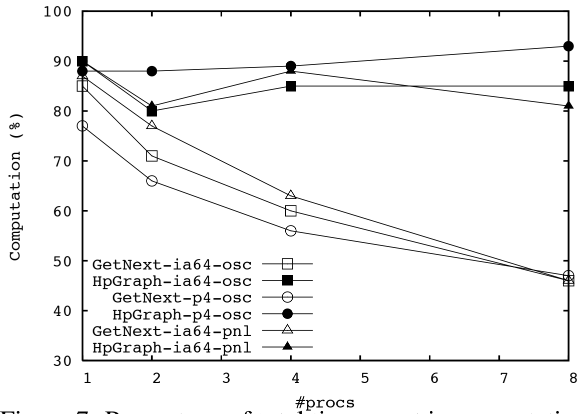

Figure 7: Percentage of total time spent in computation

7 Conclusions

In this paper, we presented a framework for automatic management of the memory-disk hierarchy in the context of block-sparse tensor contractions. A novel formulation using hypergraph partitioning was used to optimize disk I/O costs. Experimental evaluation using a sub-computation from quantum chemistry demonstrated significant improvements in disk I/O cost, overall performance, scalability, and computation efficiency.

Acknowledgments

We thank the National Science Foundation for the support of this research through grants 0121676, 0403342, and 0509467, and the U.S. Department of Energy through award DE-AC05-00OR22725. We thank the Molecular Sciences Computing Facility (MSCF) at the Pacific Northwest National Laboratory (PNNL) and the Ohio Supercomputer Center (OSC) for the use of their computing facilities.

References

AHMED, N., MATEEV, N., AND PINGALI, K. 2000. Synthesizing transformations for locality enhancement of imperfectly nested loops. In *Proc. ACM Intl. Conf. on Supercomputing*, 141–152.

BAUMGARTNER, G., BERNHOLDT, D., COCIORVA, D., HARRISON, R., HIRATA, S., LAM, C., NOOIJEN, M., PITZER, R., RAMANUJAM, J., AND SADAYAPPAN, P. 2002. A High-Level Approach to Synthesis of High-Performance Codes for Quantum Chemistry. In *Proc. of Supercomputing 2002*.

ÇATALYÜREK, U. V., AND AYKANAT, C. 1996. Decomposing irregularly sparse matrices for parallel matrix-vector multiplications. In *Proceedings of 3rd International Symposium on Solving Irregularly Structured Problems in Parallel, Irregular'96*, Springer-Verlag, vol. 1117 of *Lecture Notes in Computer Science*, 75–86.

ÇATALYÜREK, U. V., AND AYKANAT, C. 1999. Hypergraph-partitioning based decomposition for parallel spars e-matrix vector multiplication. *IEEE TPDS 10*, 7, 673–693.

CHANG, C., KURC, T., SUSSMAN, A., ÇATALYÜREK, U. V., AND SALTZ, J. 2001. A hypergraph-based workload partitioning strategy for parallel data aggregation. In *Proceedings of the Eleventh SIAM Conference on Parallel Processing for Scientific Computing*, SIAM.

CRAWFORD, T., AND III, H. S. 2000. An Introduction to Coupled Cluster Theory for Computational Chemists. In *Reviews in Computational Chemistry*, K. Lipkowitz and D. Boyd, Ed., vol. 14. John Wiley & Sons, Ltd., 33–136.

DUFF, I. S., MARRONE, M., RADICATI, G., AND VITTOLI, C. 1997. Level 3 basic linear algebra subprograms for sparse matrices: a user-level interface. *ACM Trans. Math. Softw. 23*, 3, 379–401.

HENDRICKSON, B., AND LELAND, R. 1994. The Chaco user's guide: Version 2.0. Tech. Rep. SAND94–2692, Sandia National Laboratories.

HIGH PERFORMANCE COMPUTATIONAL CHEMISTRY GROUP. 2004. *NWChem, A Computational Chemistry Package for Parallel Computers, Version 4.6*. Pacific Northwest National Laboratory.

KALÉ, L., AND KRISHNAN, S. 1993. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *Proceedings of OOPSLA'93*, ACM Press, A. Paepcke, Ed., 91–108.

KARYPIS, G., AGGRAWAL, R., KUMAR, V., AND SHEKHAR, S. 1997. Multilevel hypergraph partitioning: Applications in VLSI domain. In *Proc. of 34th Design Automation Conference*.

KHANNA, G., VYDYANATHAN, N., KURC, T., CATALYUREK, U., WYCKOFF, P., SALTZ, J., AND

SADAYAPPAN, P. 2005. A Hypergraph Partitioning Based Approach for Scheduling of Tasks with Batch-shared I/O. In *Proceedings of the 5th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2005)*. To Appear.

KODUKULA, I., AHMED, N., AND PINGALI, K. 1997. Data-centric multi-level blocking. In *Proc. SIGPLAN Conf. Programming Language Design and Implementation*, 346–357.

KRISHNAMOORTHY, S., CATALYUREK, U., NIEPLOCHA, J., ROUNTEV, A., AND SADAYAPPAN, P. 2006. An extensible global address space framework with decoupled task and data abstractions. In *Proc. IPDPS Workshop on Next Generation Software*.

KRISHNAN, S., KRISHNAMOORTHY, S., BAUMGARTNER, G., COCIORVA, D., LAM, C., SADAYAPPAN, P., RAMANUJAM, J., BERNHOLDT, D., AND CHOPPELLA, V. 2003. Data Locality Optimization for Synthesis of Efficient Out-of-Core Algorithms. In *Proc. 10th Annual International Conference on High Performance Computing (HiPC)*, Springer Verlag, 406–417.

KRISHNAN, S., KRISHNAMOORTHY, S., BAUMGARTNER, G., LAM, C.-C., RAMANUJAM, J., SADAYAPPAN, P., AND CHOPPELLA, V. 2004. Efficient synthesis of out-of-core algorithms for tensor contractions using a nonlinear optimization solver. In *The 18th International Parallel and Distributed Processing Symposium*.

LIM, A. W., AND LAM, M. S. 1998. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing 24*, 3–4 (May), 445–475.

LIM, A., LIAO, S., AND LAM, M. 2001. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *Proc. 8th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, ACM Press, 103–112.

NAVARRO, J., JUAN, A., AND LANG, T. 1994. MOB Forms: A Class of Multilevel Block Algorithms for Dense Linear Algebra Operations. In *Proc. ACM International Conference on Supercomputing*.

RANDALL, K. H. 1998. *Cilk: Efficient Multithreaded Computing*. PhD thesis, MIT Department of Electrical Engineering and Computer Science.

SAHOO, S. K., KRISHNAMOORTHY, S., PANUGANTI, R., AND SADAYAPPAN, P. 2005. Integrated loop optimizations for data locality enhancement of tensor contraction expressions. In *Proc. Supercomputing (SC 2005)*.

SALTZ, J., PONNUSAMY, R., SHARMA, S., MOON, B., AND DAS, R. 1995. A manual for the CHAOS runtime library. Tech. Rep. CS-TR-3437 and UMIACS-TR-95-34, University of Maryland, Department of Computer Science and UMIACS, March.

SINHA, A., AND KALÉ, L. 1993. A load balancing strategy for prioritized execution of tasks. In *Seventh International Parallel Processing Symposium*, 230–237.

TUMINARO, R. S., HEROUX, M., HUTCHINSON, S. A., AND SHADID, J. N. 1999. Official Aztec user's guide: Version 2.1. Tech. rep., Sandia National Laboratories.