A global address space framework for locality aware scheduling of block-sparse computations

Sriram Krishnamoorthy[1], Umit Catalyurek[2], Jarek Nieplocha[3], Atanas Rountev[1], and P. Sadayappan[1]

[1] Dept. of Computer Science and Engineering [2] Dept. of Biomedical Informatics
The Ohio State University
{krishnsr,rountev,saday}@cse.ohio-state.edu umit@bmi.osu.edu

Pacific Northwest National Laboratory
jarek.nieplocha@pnl.gov

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 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 dominant approach at present is message passing using MPI, which requires the programmer to explicitly partition the work, map it onto the parallel computer, and schedule its execution. This approach makes parallel computing very tedious and error-prone because of the myriad low-level details the programmer must contend with. One of the reasons for the continuing use of the MPI programming model is that by leaving all of the details up to the programmer, it is possible to obtain high performance — if the programmer puts in the effort to do so. But as the architecture of parallel computer systems gets more complex, the explicit orchestration of computation and communication for optimum performance will get increasingly difficult.

An emerging class of programming models require an explicit specification of partitioning and scheduling of computation onto processes, but offer a global-shared view of data, such as UPC [18], Co-Array Fortran [4], Titanium [20, 17], and Global Arrays [14, 15]. These models are generally considered easier to program than MPI because they involve only one-sided communication for data transfer, which is in some cases implicit in the program syntax, together with their global-shared view of the program data. These approaches have become prominent only recently, but are drawing a great deal of interest as alternatives to MPI as the basic parallel programming model in a variety of areas. For example, essentially all scalable parallel quantum chemistry packages use either Global Arrays or an equivalent [7, 19, 12, 13, 16, 6].

For computations involving data structures more general than dense matrices, performing completely automatic program transformations to parallelize them and optimize their execution time is a challenging task. We propose a model in which the data and computation abstractions are geared towards expressing and exploiting the locality inherent in the problem. The data abstraction encapsulates the locality of access. The user expresses the parallelism in the computation in the form of a *task pool*, which is then used by the runtime system to schedule the computation to maximize locality. The decoupling of the computation and data abstractions enables the implementation of efficient operations on existing data structures, extending their capabilities. We

1-4244-0910-1/07/$20.00 ©2007 IEEE.

present our experiences with this approach in the context of computations on multi-dimensional block-sparse arrays. The motivating applications are presented in Section 2. The locality-aware abstractions are discussed in Section 3. The hypergraph partitioning problem, employed in communication and disk I/O minimization, is described in 4. Section 5 discusses our approach to minimizing communication for in-memory computations. Section 6 discusses disk I/O minimization for out-of-core computations. Section 7 concludes the paper.

2 Motivation

One of the major motivations for the development of the proposed primitives is our work on the Tensor Contraction Engine (TCE) [2] synthesis system. TCE 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. An operation on the indices of the segments that form a block of an array determines if that block is non-zero. The wide-ranging sizes of the blocks lead to significant variations in the computation and communication times involved in processing a block. The large sizes of the arrays can significantly increase communication costs, if locality is not taken into account. For large systems the data can be too large to fit in the collective physical memory in the parallel system. This necessitates the schedules of disk I/O to ensure that the data is available in memory when the computation is being performed.

In general, applications with the following characteristics can benefit from our proposed scheme:

- Can be partitioned into independent tasks,
- Involve many more tasks than the number of processors,
- Have wide variation in task execution times, and
- Operate on coarse-grain data, and incur communication costs if the task and the data it operates on are not co-located.
- Data is distributed in the collective physical memories of the parallel system or in secondary storage.

3 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 [10].

3.1 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 global memory and disk.

3.2 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.

4 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

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

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

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

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 [11]. The two relevant ones for our context are cut-net and connectivity-1, defined as follows:

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

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

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 multi-level paradigm, such as hMETIS [9] and PaToH [3], have been shown to compute good partitions quickly for this NP-hard problem.

5 In-Memory Computations

A computation, consisting of a set of independent tasks, 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. 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 all the tasks assigned to it.

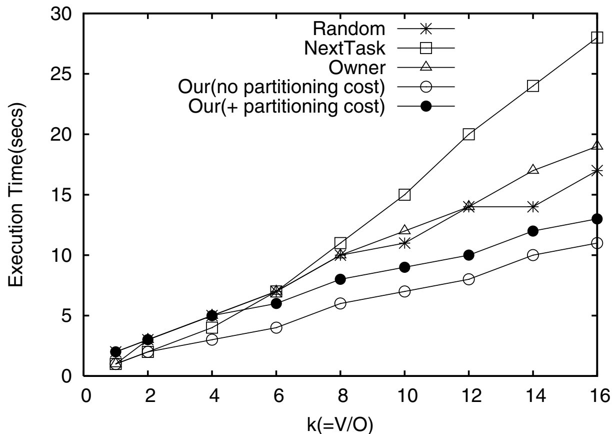

Figure 1. Execution time of block-sparse matrix multiply (32 processors)

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.

5.1 Communication Minimization

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 prespecified 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 2.

5.2 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 *Random* scheme, each processor traverses the entire list of tasks in the same order, and randomly decides whether it is to be executed by it. Since all the processors start with the same random seed, they all generate the same sequence of pseudo-random numbers. 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 *Owner* 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 ensures locality for the array used to determine the ownership.

The *NextTask* scheme employs dynamic load balancing. 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 [1]. This is also the typical model of parallelization used in many applications, including some quantum chemistry codes [8].

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

$$a, b, c, d : O$$
$$i : V$$
$$C[a, b, c, d] = A[a, b, i] * B[i, c, d]$$

where O and V correspond to the number of occupied and virtual orbitals, respectively. The O index was set at 160 with four symmetry segments of lengths 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 execution times for the different schemes for 32 processors are shown in Fig. 1. For our approach (labeled *Our*) the cost is shown including and excluding the overhead of hypergraph partitioning. Our approach shows im-

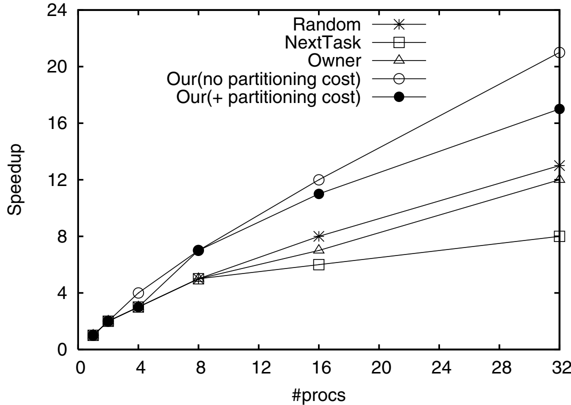

Figure 2. Scalability of block-sparse matrix multiply for k=(V/O)=16

proved performance even after including the overhead of hypergraph partitioning.

The speedups obtained by the different schemes, for k being 16, are shown in Fig. 2. Our approach achieves a speed-up of up to 20 on 32 processors, excluding partitioning cost. Note that we did not exploit overlap of communication with computation.

6 Out-of-core Computations

In this section, we consider the scheduling problem in the context of data in secondary storage, in a parallel file system or distributed amongst the local disks of processors. The objective is to determine a computation schedule, so as to minimize the total disk I/O cost. 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.

6.1 One-Level Partitioning

In this section, we describe a direct application of hypergraph partitioning to the disk I/O minimization problem. 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, modeled as 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. 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 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 the tasks can be done by employing the scheme described in the previous section. There is no reuse of data across the different stages. Thus, a reduction in the number of stages is generally beneficial.

6.2 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. We present an alternative use of hypergraph partitioning, referred to 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 ex-

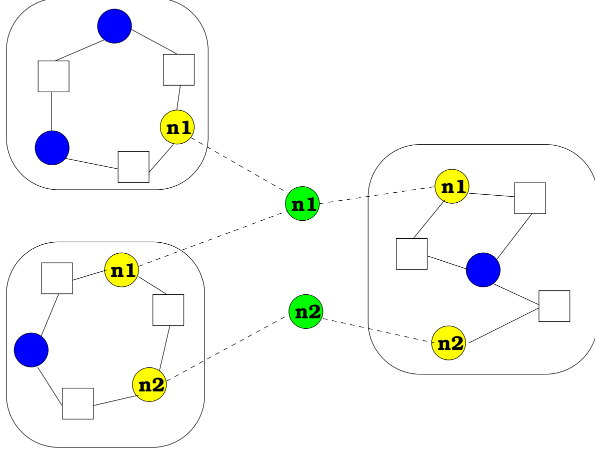

Figure 3. Illustration of one-level partitioning

ceed 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.

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.

6.3 Integrated Approach

The integrated algorithm, referred to as *two-level partitioning* 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, 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, 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. 4 shows a possible partitioning of the same computation as in Fig. 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 Fig. 3. 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. 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.

6.4 Experimental Evaluation

We evaluate our approach using the following Coupled Cluster Doubles (CCD) [5] sub-computation:

$p3, p4, p5, p7 : V$
$h1, h2, h6, h8 : O$
input-output arrays : $i0, t, v1, v2$
intermediate arrays : $i1$
$i1[h6, p3, h1, p5] += v1[h6, p3, h1, p5]$
$i1[h6, p3, h1, p5] += t[p3, p7, h1, h8] * v2[h6, h8, p5, p7]$
$i0[p3, p4, h1, h2] += t[p3, p5, h1, h6] * i1[h6, p4, h2, p5]$

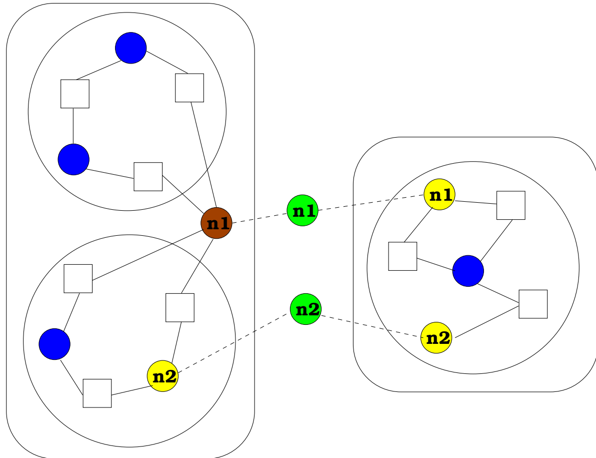

Figure 4. Illustration of two-level partitioning

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

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 [7]. 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, similar to the *NextTask* scheme described earlier, is used to distribute the computation amongst the processors. 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 by 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. This is the system used in the evaluation of our in-memory optimization described earlier, but with a different 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-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 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 improvements obtained would be even higher if the cost of replicating the input arrays is included in the execution time of GetNext.

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

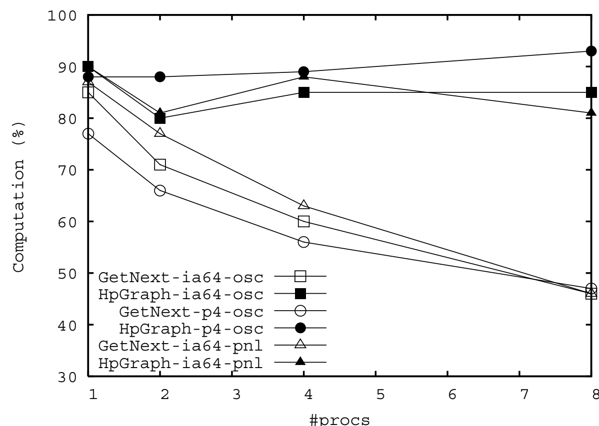

Figure 5. Percentage of time in computation

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. 5. It shows the consistent high efficiency achieved by HpGraph, despite the additional overhead of hypergraph partitioning.

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

[1] Openmp specification. http://www.openmp.org/specs.

[2] G. Baumgartner, D. Bernholdt, D. Cociorva, R. Harrison, S. Hirata, C. Lam, M. Nooijen, R. Pitzer, J. Ramanujam, and P. Sadayappan. A High-Level Approach to Synthesis of High-Performance Codes for Quantum Chemistry. In *Proc. of Supercomputing 2002*, November 2002.

[3] U. V. Çatalyürek and C. Aykanat. Hypergraph-partitioning based decomposition for parallel spars e-matrix vector multiplication. *IEEE TPDS*, 10(7):673–693, 1999.

[4] Co-array fortran. http://www.co-array.org/.

[5] T. Crawford and H. S. 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.

[6] M. Guest. Computing for science. http://www.dl.ac.uk/CFS/cfs.html, 8 March 2004.

[7] High Performance Computational Chemistry Group. *NWChem, A Computational Chemistry Package for Parallel Computers, Version 4.6*. Pacific Northwest National Laboratory, 2004.

[8] S. Hitara. Tensor contraction engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories. 107(46):9887–9897, 2003.

[9] G. Karypis, R. Aggrawal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: Applications in VLSI domain. In *Proc. of 34th Design Automation Conference*, 1997.

[10] S. Krishnamoorthy, U. Catalyurek, J. Nieplocha, A. Rountev, and P. Sadayappan. An extensible global address space framework with decoupled task and data abstractions. In *Proc. IPDPS Workshop on Next Generation Software*, 2006.

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

[12] H. Lischka and T. Mueller. The columbus parallel CI program project. http://www.itc.univie.ac.at/~hans/Columbus/columbus_parallel.html, May 6 2004.

[13] Molcas 6. http://www.teokem.lu.se/molcas/.

[14] J. Nieplocha and R. Harrison. Shared-memory programming in metacomputing environments: The global array approach. *The Journal of Supercomputing*, 11:119–136, 1997.

[15] J. Nieplocha, R. Harrison, and R. Littlefield. Global Arrays: A portable shared memory model for distributed memory computers. In *Proc. Supercomputing '94*, pages 340–349, 1994.

[16] Q-Chem, Inc. Q-chem. http://www.q-chem.com/, May 13 2004.

[17] Titanium. http://www.cs.berkeley.edu/projects/titanium/.

[18] Unified parallel c. http://upc.nersc.gov/ and http://upc.gwu.edu/.

[19] H. J. Werner, P. J. Knowles, R. Lindh, M. Schütz, P. Celani, T. Korona, F. R. Manby, G. Rauhut, R. D. Amos, A. Bernhardsson, A. Berning, D. L. Cooper, M. J. O. Deegan, A. J. Dobbyn, F. Eckert, C. Hampel, G. Hetzer, A. W. Lloyd, S. J. McNicholas, W. Meyer, M. E. Mura, A. Nicklass, P. Palmieri, R. Pitzer, U. Schumann, H. Stoll, A. J. Stone, R. Tarroni, and T. Thorsteinsson. Molpro, version 2002.6, a package of ab initio programs. http://www.molpro.net, 2003.

[20] K. Yelick, L. Semenzato, G. Pike, C. Miyamuto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance java dialect. *Concurrency: Practice and Experience*, 10(11–13), 1998.