An Approach to Locality-Conscious Load Balancing and Transparent Memory Hierarchy Management with a Global-Address-Space Parallel Programming Model

Sriram Krishnamoorthy[1], Umit Catalyurek[2], Jarek Nieplocha[3], 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

The development of efficient parallel out-of-core applications is often tedious, because of the need to explicitly manage the movement of data between files and data structures of the parallel program. Several large-scale applications require multiple passes of processing over data too large to fit in memory, where significant concurrency exists within each pass. This paper describes a global-address-space framework for the convenient specification and efficient execution of parallel out-of-core applications operating on block-sparse data. The programming model provides a global view of block-sparse matrices and a mechanism for the expression of parallel tasks that operate on block-sparse data. The tasks are automatically partitioned into phases that operate on memory-resident data, and mapped onto processors to optimize load balance and data locality. Experimental results are presented that demonstrate the utility of the approach.

1 Introduction

The development of scalable parallel application codes is a challenging task. Global-address-space parallel programming models like Co-Array Fortran, Unified Parallel C (UPC) and Global Arrays (GA) have sought to raise the level of abstraction from distributed memory message-passing models like MPI, without sacrificing performance. While simpler than distributed memory models, prevalent global address space models still require varying amounts of information from the user in terms of data distribution and computation partitioning. The models are not completely process-oblivious. Further, these models do not provide a unified treatment of other levels of the memory hierarchy, notably the handling of secondary storage. Thus, the development of parallel "out-of-core" applications remains a very tedious and challenging task.

A parallel out-of-core application typically makes multiple passes over disk-resident data. During each pass, a portion of the data is read in from disk, computed on, and then written back to disk. For parallel computation, the data brought in from disk must be partitioned and distributed among the processors, and possibly communicated between processors during the computation. The need for explicit orchestration of all the data movement makes the task of application development very tedious. The Global Arrays (GA) [21] and Disk Resident Arrays (DRA) [20] libraries facilitate the development of parallel out-of-core applications using dense multi-dimensional arrays, by providing a global shared view of multi-dimensional arrays in memory and disk, respectively. We use this as a basis, to develop an enhanced framework to support the development of applications that operate on block-sparse matrices.

In this paper, we present an approach that provides a global unified view of the memory-disk hierarchy to the user. The user specifies the data in terms of fundamental units ("bricks") in a global data collection, and computation as tasks operating on these units, without any restriction on the total size of the data collection relative to physical memory size, or any explicit specification of data movement between secondary storage and primary memory. The information provided by the user is used to determine a data and computation partitioning aimed at minimizing inter-processor communication and disk I/O, while achieving good computational load-balance. This is achieved by generation of a hypergraph capturing task-data relationships, that is then partitioned through two passes of a hypergraph partitioner - an "outer" step that partitions the set of tasks into groups so that their data can fit within memory, and an "inner" step that maps tasks to processors.

We have developed a prototype implementation of the

1-4244-0054-6/06/$20.00

model over the ARMCI (Aggregate Remote Memory Copy Interface) [19] and GA suite. We demonstrate the use of this framework in the context of the Tensor Contraction Engine [2], a domain-specific compiler targeted at a class of ab initio quantum chemistry computations. The computations involve tensor contractions with block-sparse tensors that might be too large to fit into the collective memory of the processors. The data is specified in terms of elementary multi-dimensional bricks into which the block-sparse tensors are partitioned. The movement of data between disk and physical memory, and the mapping of computation amongst the processes is determined automatically. We present results that show that such user-transparent mechanisms can achieve good performance, while extending the computational domain effectively handled by extant global address space programming models.

The paper is organized as follows. In Section 2, we discuss the applications that motivated our work. The Global Arrays suite is described in Section 3.1 and the proposed enhancements in Section 3.2. The data and computation abstractions to manipulate block-sparse arrays are introduced in Section 4. The implementation of the framework using a hypergraph partitioner is discussed in Section 5. Experimental results are discussed in Section 6. Related work is detailed in Section 7. Section 8 concludes the paper.

2 Target Applications

In this section, we list two target applications that motivated the proposed framework. In general, the developed framework mapping can benefit applications that:

- 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 disk I/O costs and/or inter-processor communication costs if the task and the data it operates on are not co-located.

Computations with data dependences can also benefit from this mechanism, provided there is enough parallelism at any point in the computation. For example, while performing a sequence of block-sparse matrix multiplies, each matrix multiply can be treated as a set of independent tasks and processed using this mechanism.

2.1 Tensor Contraction Expressions

The development of the framework was primarily motivated by our work on the Tensor Contraction Engine (TCE) [3] synthesis system. The 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 it is non-zero. The wide-ranging sizes of the blocks leads to significant variation 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. The arrays can also be too large to fit in the aggregate physical memory of a parallel machine, thereby requiring out-of-core treatment of the data.

2.2 Lennard Jones Energy Minimization Using Force Decomposition

Load balancing is important for force decomposition molecular dynamics algorithms. The array of forces of dimension $N \times N$ is divided into multiple blocks of size $m \times m$, where m is the block size and N is the total number of atoms. Each process owns N/P atoms, where P is the total number of processors, and each processor computes a fixed subset of inter-atomic forces [24]. The forces between atoms farther from each other than the *cut-off distance* need not be evaluated, resulting in unequal processing times for each subset of the force-matrix. This, together with the block decomposition of the force matrix, leads to load imbalance.

3 Overview of Approach

3.1 Global Arrays Programming Suite

The Global Arrays programming suite [22] 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 [19] 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 [9], GPSHMEM [23], and Global Arrays.

The next higher level is the Global Arrays (GA) library. GA provides 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 [14] (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 [22].

The Disk Resident Arrays (DRA) model [20] 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.

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.

3.2 Proposed Enhancements

The GA/DRA framework is convenient to develop parallel out-of-core applications that operate on large dense multidimensional arrays. However, for applications operating on block-sparse data, with the current GA/DRA framework, a multidimensional block-sparse array must either be represented as multiple arrays, one per nonzero block, or as a compacted one-dimensional array that stores a sequence of linearized blocks. With the Tensor Contraction Engine, a large number of multidimensional tensor contractions are required, where each tensor is block-sparse, with widely varying sizes for the nonzero blocks. A tensor contraction is essentially a generalized multidimensional matrix-matrix multiplication. With block-sparse tensors, a contraction requires pairwise contractions of several nonzero block pairs. In this context, achieving computational load balance among the processors, along with minimization of inter-processor communication is not straightforward. Further, the block-sparse tensors could be too large to fit in the aggregate physical memory of the parallel machine. Developing efficient parallel out-of-core algorithms for such block-sparse tensors is very tedious.

The framework described in this paper was motivated by the difficulty of developing efficient parallel out-of-core applications operating on block-sparse data structures. The framework comprises of the following:

- A globally addressable "bricked" representation of block-sparse multidimensional arrays. Each non-zero block is represented as a set of multi-dimensional "bricks", where the data layout within a brick conforms to a standard "row-major" ordering. The bricks are globally addressable using a multi-dimensional tuple. The total size of the collection of bricks of a block-sparse array is not restricted by the aggregate physical memory of the parallel machine. The movement of bricks between disk and main memory is transparent to an application program.
- "Task pools" for specification of sets of concurrent tasks that operate on bricks of the globally addressable block-sparse arrays. Each task in a task pool specifies source and result bricks. The mapping of tasks to processors, and the movement of input data bricks to a task's processor is implemented by the system, with the goal of load-balanced execution with minimization of disk-to-memory data movement and inter-processor communication.
- Automatic mapping and scheduling of the tasks in a task pool onto processors, along with the generation of disk I/O and inter-processor communication needed to achieve parallel out-of-core execution of the collection of tasks.

4 Locality-aware Abstractions

In this section, we briefly discuss the data and computation abstractions to manipulate block-sparse matrices. 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 [17].

4.1 Abstraction for Block-Sparse Matrices

The abstraction for multi-dimensional block-sparse matrices provides collective functions for creating and destroying arrays and non-collective functions to get/put data from/to the distributed block-sparse array.

The non-zero blocks of the array are divided into bricks of a specified 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.

A replicated index is created is store information pertaining to the distribution of the non-zero bricks in block-sparse array. The one-sided mechanisms provided by the ARMCI library, together with the replicated index, enables the non-collective access to arbitrary bricks in the array.

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, a function that takes as argument the block indices and returns whether it is zero, can be provided.

4.2 Computation Specification

The computation abstraction provided to the user enables the specification of a set of independent tasks to be executed in parallel. For each such set, all processes collectively create a task pool object.

Each task in the task pool is identified by the routine to be invoked to process that task, identified by a function handle, and the set of *locality elements* it operates upon. In addition, any private data specific to that task can also be specified. Each locality element corresponds to a global data region, identified by its global address, size, and its access mode. Three access modes are supported. Read, write, and update accessing modes allow for put, get, and accumulate of global data, respectively.

All processes populate the task pool before *sealing* it. Once a task pool is sealed no more tasks can be added to it. At this point the task pool is fully defined and any start-time optimizations can be performed.

Subsequently, all the processes collectively process the tasks in the task pool. A task pool, once created, can be processed multiple times, thus amortizing the cost of any start-time optimizations.

5 Implementation

In this section, we discuss the implementation of the framework. The computation in a task pool is a collection of tasks, where each task explicitly specifies its operand data bricks. We modeled the dependencies of tasks to its operand data bricks using the computational hypergraph model [5]. In the constructed hypergraph, vertices represent tasks and hyper-edges represent the operand data bricks. Weight of a task and cost of an hyper-edge reflects the relative estimated execution time of the task, and the size of the operand data brick, to characterize the cost of moving the brick, respectively.

Here, we propose a hypergraph partitioning-based two-level approach for the generation of a schedule for disk I/O operations, inter-processor communication operations and task execution:

- The "outer" level of partitioning divides the task pool into multiple disjoint *clusters*, sets of tasks, where total amount memory required by each of these clusters is less than the total collective memory of the parallel machine. Then each one of these clusters is executed in the parallel machine one by one. An attempt is made to minimize the number of such clusters, as well as to reduce the I/O overhead due to data bricks shared among multiple clusters.
- The "inner" level of partitioning takes as input a cluster from the outer-level partitioning, and computes a mapping of task (and possibly data) to processors in order to optimize the execution of the cluster. The execution of a cluster requires the loading of operand data bricks from disk to memory, parallel execution of the cluster's tasks among the processors (possibly interspersed by inter-processor communication), and write-back of result data bricks. Hence, mapping is achieved by performing a balanced P-way partitioning of the cluster's hypergraph where the total hyper-edge cut is minimized.

We start with some preliminary definitions for hypergraphs. Then we present the outer-level and inner-level partitioning phases in more detail.

5.1 Hypergraph Partitioning

A hypergraph $H = (V,N)$ is defined as a set of vertices V and a set of nets (hyper-edges) N among those vertices. For every net n_j, s_j is equal to the number of its vertices, i.e., $s_j = |n_j|$. 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. A P-way partition $\Pi = \{V_1, V_2, ..., V_P\}$ of H is a partitioning of vertices of H that satisfies the following criteria: 1) each part is a nonempty subset of V , 2) the parts are pairwise disjoint, and 3) the union of the P parts is equal to V . In the traditional hypergraph partitioning problem, a partition is said to be balanced if $W_p \leq W_{avg}(1+\varepsilon)$ for $1 \leq p \leq P$, where $Wp = \sum_{v_i \in V_p} w_i$ is the sum of the vertex weights of part V_p , $W_{avg} = (\sum_{v_i \in V} w_i)/P$ denotes the weight of each part under the perfect load balance condition, and ε represents the predetermined maximum imbalance ratio allowed. In a partition P 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 cut if it connects more than one part (i.e. $\lambda_j > 1$). The cut nets are also referred to as external nets, denoted as N_E. There are various cut-size definitions for representing $\chi(\Pi)$ of a partition P. The relevant, connectivity-1 definition is:

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

In equation 1, 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 among the part weights is maintained. Algorithms based on the multi-level paradigm, such as hMETIS [16] and PaToH [5], have been shown to compute good partitions quickly for this NP-hard problem.

In this work we define a new hypergraph partitioning problem that we call *Bounded Incident Net Weight (BINW) Partitioning* Let $\Pi = \{V_1, V_2, ..., V_P\}$ be the P-way partition of hypergraph H. The cost of a partition is again computed using the connectivity-1 cut-size definition (Eq.1), but the constraint on the partitioning is different. Let $I(V_i)$ denote the nets that are incident on vertices in (V_i), i.e., $I(V_i) = \{n_j | v_k \in n_j, \forall v_k \in V_i\}$. The BINW partitioning is defined as finding a minimum cost partition where each part's incident net weight sum is bounded by a predetermined weight constraint M:

$$\sum_{n_j \in I(V_i)} c(n_j) \leq M \quad (2)$$

Please note that P is not predetermined in this problem, however, minimizing the connectivity-1 cost while obeying the incident net weight constraint would also minimize the number of parts.

5.2 Outer-level Partitioning: Out-of-Core Partitioning

The goal of the outer-level partitioning is to divide the tasks into clusters such that the total amount of memory required by the tasks of each cluster is less than available aggregate memory. In order to achieve this we use the computational hypergraph model together with Bounded Incident Net Weight Partitioning described in the previous section. Consider a P-way BINW partitioning Π of a computational hypergraph H, where the weight constraint M is set to the aggregate memory of the parallel machine. Since the data bricks that are required by each task is represented by nets connected to that task, the data bricks required by the tasks of a part P_i constitute the incident net set $I(V_i)$. By definition of BINW partitioning, we know that the total sum of the net costs (size of the data elements) is less than M. Hence, all data bricks required by a cluster of tasks will fit into the aggregate memory of the parallel machine. Each data brick needs to be read/written at least once for each cluster. Minimizing the connectivity-1 metric corresponds to minimizing the number of times that a data brick is shared among the clusters - hence it corresponds to minimizing the I/O due to data bricks shared among the clusters.

One of the challenges in BINW partitioning is that it is not possible to directly determine the smallest feasible P. One approach could be to estimate the number of parts, and partition using an existing hypergraph partitioner that uses the traditional vertex weight balance constraint, and check whether the resulting partition violates the memory constraint. If it produces an infeasible partition, the number of parts can be increased and tried again. This approach has two problems. First, since we will be forcing a partition into a predetermined number of parts without any control on the actual incident net weights, it is very possible that it will result in a high load imbalance in incident net weights, resulting in under-utilization of the parallel machine's memory. The second problem is that finding a solution will likely require multiple partitioning, increasing the time to generate the partition.

We have developed a BINW partitioner by modifying the successful serial hypergraph partitioner PaToH [6]. Since PaToH achieves P-way partitioning through recursive bisection, we have chosen to use the same framework. During the recursive bisection, the nets that are in the cut are split in order to achieve correct accounting of the connectivity-1 cost metric. In PaToH, the default action for the size-1 split-nets is to discard them, since they cannot be in the cut for a future bisection. However, since our weight constraint is based on incident net weights, we have modified the code so that the sum of the weights of such size-1 nets are accumulated in a separate weight variable for each vertex. While computing the weight constraint, those weights are aggregated with the sum of the internal net weights to compute a part's incident net weight.

We have also modified the stopping condition of the recursive function. Since P is not predetermined, we stop when a incident net weight of a part is less than or equal to the predetermined weight constraint M.

5.3 Inner-level Partitioning: Locality-aware Load-balancing

The goal of the inner-level partitioning is to compute a mapping of tasks and data bricks to the processors of the parallel machine for an optimized execution.

When a task is assigned to a processor, the input data bricks 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 bricks 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 data across tasks.

Hence 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.

We model the problem of locality-aware load-balancing as a hypergraph partitioning problem. Again each task and data brick is represented, respectively, by a vertex and net in the hypergraph. The computational load of tasks and the size of data bricks are used as weights and costs of respective vertices and nets. Additionally, each data brick is also represented by a zero-weight vertex. For each data brick, the corresponding net connects the vertices corresponding to it and the tasks that access it.

Consider a P-way partition Π of such a hypergraph. We decode the partitioning as follows. For each $v_i \in V_i$, we map the corresponding task/data-brick to processor P_i. Cut nets corresponds to the data bricks that are shared and hence required by multiple processors. Those data bricks are shared by λ_j parts, they need to be communicated exactly $\lambda_j - 1$ times, that is, moved from the owner processor (the processor to which the data-brick vertex is assigned) to all the processors that need it, except itself. Hence, by assigning the size of data bricks as net costs, the proposed method reduces the task and data mapping problem to the P-way hypergraph partitioning problem using the *connectivity-1* cost function (Eq 1).

6 Experimental Results

In this section, we provide preliminary performance data on the prototype implementation of the framework. Experiments were performed to evaluate the primitives on the Colony2a system in the Pacific Northwest National Laboratory, a twenty-four node cluster with each node being a dual 1GHz Itanium-2 with 6GB memory, interconnected using Myrinet.

We first evaluate the utility of the approach in abstracting away explicit disk I/O handling, by comparing it with an alternative implementation based on virtual memory. Virtual memory provides an address space potentially much larger than the available physical memory. The virtual memory mechanism provides for transparent movement of data between disk and main memory.

We evaluated the approaches using dense matrix-matrix multiply as shown below:

$C[O,O] = A[O,V] * B[V,O]$

O was set to 1000, while V was varied between 100,000 and a million.

For the implementation using virtual memory, referred to as *vm-layout* the data is organized in exactly the same way as for our hypergraph-based partitioning approach, i.e. as a collection of bricks, with the elements of a brick being stored in row-major order. The computation of a partial product using a brick-brick interaction is done using *DGEMM*. The results are shown in Table 1.

V	Brick size	vm-layout	Hypergraph	
			Partitioning time	Total time
100K	500	85	0.4	89
100K	1000	86	< 0.1	87
500K	500	958	2.0	578
500K	1000	710	0.1	532
1M	500	5166	6.0	1191
1M	1000	1960	0.3	1108

Table 1. Execution times of dense matrix multiplication, in seconds

We show results of experiments with two brick sizes (500 and 1000), and matrices of three sizes (A's size: 1000 by 100K; 1000 by 500K; 1000 by 1M). For the smallest case, all three matrices can fit in physical memory, and the performance of the vm-based implementation and the hypergraph-partitioning-based implementation are comparable. With the medium-sized matrix (1000 by 500K), the total size of the three matrices exceeds the 6 Gbytes of physical memory on the machine. The vm-based implementation is clearly slower than the hypergraph-partitioning-based approach. With the large matrix (1000 by 1M), the vm-based approach is again slower, with the difference in performance between the two being greater. Performance for the medium and large case is better with a brick-size of 1000 than with brick-size of 500. The hypergraph partitioning time also decreases with increasing brick size, since the number of bricks (nets) and number of vertices decreases. The total hypergraph partitioning time is quite negligible for all the cases.

We next present performance data for a block-sparse tensor contraction example:

C[O,O,V,O] = A[O,O,V,V] * B[V,O]

C and A are 4-dimensional block-sparse tensors, while B is a 2-dimensional block-sparse tensor. High-accuracy quantum chemistry models such as the coupled cluster models [10] are replete with such block-sparse tensor contractions. Each array index ranges either over occupied electron orbitals (O) or over virtual orbitals (V). The index ranges O and V are partitioned into a fixed (power-of-two) number of so-called "irreps". The number of irreps depends on the nature of spatial symmetry of the chemical molecule being modeled, and is often 4 or 8. For the experiments below, O consisted of 4 spatial symmetry blocks of sizes 100, 50, 25, and 25. V was twice O, with symmetry blocks of size 200, 100, 50, and 50. A block (p,q,r,s) of multidimensional tensor C is nonzero iff Exclusive_Or(p,q,r,s) equals zero. For 2-D tensor B, only the four diagonal blocks are non-zero, and

Brick size	No. Processors		
	1	2	4
25	667	464	246
50	599	370	226
75	638	349	300
100	660	466	321

Table 2. Time for parallel out-of-core block-sparse constraction, in seconds.

for the 4-D tensors, 64 of the 256 blocks are nonzero. For example, block (1,2,0,3), (1,2,1,2), (1,2,2,1) and (1,2,3,0) are nonzero. It can be observed that the range of sizes of nonzero blocks varies considerably. For A, the largest block has 400,000,000 nonzeros while the smallest nonzero block has 1,562,500 nonzeros.

The cost was evaluated for brick sizes varied between 25 and 100. The execution times are shown in Table 2.

The scalability of the initial prototype is being improved through a number of implementation optimizations (e.g. collective memory allocation for multiple blocks rather than one at a time).

7 Related Work

The use of recursive and blocked data layout for dense matrix algorithms has been the subject of many studies [8, 12]. Abstractions for block-sparse matrices also exist in the context of linear algebra and iterative solvers [11]. Aztec [27] is a parallel iterative solver package that provides a global view of a distributed matrix. Advanced partitioning techniques [13] 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 for locality-aware load-balancing are not tightly coupled with block-sparse matrices, and can be utilized in a wide range of contexts.

The use of hierarchically tiled arrays as a primitive data type has been proposed and evaluated in the context of MATLAB [1, 4].

Dynamic load-balancing based on work-stealing has been studied, particularly for state-space search [26] . Charm++ [15] supports dynamic load-balancing by object migration. Cilk [25] supports load-balancing of computations based on work-stealing. OpenMP exploits parallelism at the loop level by distributing different iterations to different processors. Locality is not taken into consideration in any of these schemes.

Çatalyürek and Aykanat [5] have used hypergraph-partitioning to parallelize sparse matrix-vector multiplications. Chang et al. [7] performed parallel data aggregation based on hypergraphs.

Thus, different aspects of the proposed framework bear similarities to previous work, but we are unaware of any other work that has combined all three of the following facets of the framework developed here:

- Implementation of a globally addressable "brick" repository, and bricked representation of block-sparse matrices
- Automatic deduction of data locality relations from a high-level specification of the computation
- Automatic synthesis of schedule for out-of-core parallel execution of block-sparse matrix computations through runtime optimization

The work presented in this paper represents an extension of our work [18] that addressed dat aanc computation abstractions for "in-core" block-sparse computations.

8 Conclusions

The paper describes the design and implementation of high-level abstractions for specifying parallel computations on 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 automatically groups the tasks into a sequence of phases, and determines the computation mapping and data movement for tasks in each phase. Preliminary experimental results demonstrate that the approach is very promising.

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) for the use of their computing facilities.

References

[1] G. Almasi, L. D. Rose, B. B. Fraguela, J. Moreira, and D. Padua. Programming for locality and parallelism with hierarchically tiled arrays. In *Proc. 16th Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2003.

[2] G. Baumgartner, A. Auer, D. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C. Lam, Q. Lu, M. Nooijen, R. 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, 2005.

[3] 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.

[4] G. Bikshandi, B. B. Fraguela, J. Guo, M. J. Garzaran, G. Almasi, J. Moreira, and D. Padua. Implementation of Parallel Numerical Algorithms Using Hierarchically Tiled Arrays. In *Proc. 17th Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2004.

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

[6] U. V. Çatalyürek and C. Aykanat. *PaToH: A Multilevel Hypergraph Partitioning Tool, Version 3.0*. Bilkent University, Department of Computer Engineering, 1999.

[7] C. Chang, T. Kurc, A. Sussman, U. V. Çatalyürek, and J. Saltz. A hypergraph-based workload partitioning strategy for parallel data aggregation. In *Proceedings of the Eleventh SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, March 2001.

[8] S. Chatterjee, A. Lebeck, P. Patnala, and M. Thottethodi. Recursive array layouts and fast matrix multiplication. *IEEE Trans. Parallel and Distributed Systems*, 2002.

[9] C. Coarfa, Y. Dotsenko, and J. Mellor-Crummey. A Multi-Platform Co-Array Fortran Compiler. In *Proc. of PACT*, 2004.

[10] 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.

[11] I. S. Duff, M. Marrone, G. Radicati, and C. Vittoli. Level 3 basic linear algebra subprograms for sparse matrices: a user-level interface. *ACM Trans. Math. Softw.*, 23(3):379–401, 1997.

[12] E. Elmroth, F. Gustavson, I. Jonsson, and B. Kagstrom. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Review*, 2004.

[13] B. Hendrickson and R. Leland. The Chaco user's guide: Version 2.0. Technical Report SAND94–2692, Sandia National Laboratories, 1994.

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

[15] L. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

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

[17] 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.

[18] S. Krishnamoorthy, J. Nieplocha, and P. Sadayappan. Data and computation abstractions for dynamic and irregular computations. In *Proc. Intl. Conference on High Performance Computing*, 2005.

[19] J. Nieplocha and B. Carpenter. ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems. In *Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP)*, 1999.

[20] J. Nieplocha and I. Foster. Disk Resident Arrays: An Array-Oriented I/O Library for Out-Of-Core Computations. In *Proc. 6th Symposium on the Frontiers of Massively Parallel Computation*, pages 196–204, 1996.

[21] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: a portable programming model for distributed memory computers. In *Supercomputing*, pages 340–349, 1994.

[22] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apra. Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit. *Intern. J. High Perf. Comp. Applications*, to appear, 2005.

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

[24] S. J. Plimpton and B. A. Hendrickson. Parallel molecular dynamics with the embedded atom method. In *Proc. of Materials Theory and Modelling*, page 37. MRS Proceedings, 1993.

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

[26] A. Sinha and L. Kalé. A load balancing strategy for prioritized execution of tasks. In *Seventh International Parallel Processing Symposium*, pages 230–237, Newport Beach, CA., April 1993.

[27] R. S. Tuminaro, M. Heroux, S. A. Hutchinson, and J. N. Shadid. Official Aztec user's guide: Version 2.1. Technical report, Sandia National Laboratories, 1999.