

Locality-aware Load Balancing for Dynamic and Irregular Computations

Sriram Krishnamoorthy¹ P Sadayappan¹ Jarek Nieplocha² Manoj Krishnan²

¹Department of Computer and Information Science
The Ohio State University
Columbus, OH, USA - 43210
{krishnsr, saday}@cis.ohio-state.edu

²Computational Sciences and Mathematics
Pacific Northwest National Laboratory, Richland, WA 99352, USA
{jarek.nieplocha, manojkumar.krishnan}@pnl.gov

1 Introduction

Development of scalable application codes is a challenging task. It requires an understanding of system architecture, programming model, and ability to identify and express parallelism in the underlying problem. Design patterns can help programmers by presenting a set of recipes for expressing and implementing parallelism in the context of a particular programming model and classes of algorithms.

In the current paper, we consider the Global Arrays shared memory programming model. In particular, we focus on a relatively common design pattern associated with implementations of dynamic load balancing in GA programs. Dynamic load balancing is often used in dynamic and irregular problems that have been parallelized using GA, in different application areas such as the electronic structure, molecular dynamics, or computational physics based on adaptive mesh refinement. The global/shared view of data and efficient one-sided communication offered by the GA toolkit facilitate a relatively straightforward implementation of dynamic load balancing, as compared to MPI. In particular, the programmer can take advantage of atomic operations such as *fetch-and-add* to schedule the execution of parallel tasks and rely on one-sided access to shared data to implement communication involved in executing individual tasks, regardless of the data ownership. For example, a common design pattern used in several computational chemistry algorithms uses an atomic update of a shared variable that corresponds to the current task index, pointing to either an explicit or implicit task list: a *get* for accessing the required sections of global arrays, followed by calculation, and then updating the result array using a *put* or *atomic accumulate* operation. The decoupling of the underlying data placement from computation makes the implementation of algorithms relying on dynamic load balancing relatively simple to even a non-expert users. However, it does not address the problem of efficient scheduling of tasks, and locality optimizations that are required to maximize performance, taking into account the architectural details such as network topology, latency and bandwidth, and other factors such as the communication contention in access to the required data.

The computational structure in these problems is often expressible as directed acyclic graphs (DAG). Each node in the graph represents a task or a work element and edges represent input-output relationships. Each node can be executed only after all nodes with edges incident on it have been processed. When there are independent paths through the graph, the computation is said to exhibit *inter-task parallelism*, and more than one task can be executed in parallel. In this paper, we propose a work-sharing construct to schedule the DAG on a parallel system, to minimize the overall execution time.

We consider two work-sharing models. In the static work-sharing model, the DAG is completely specified before it is processed. Alternatively, the dynamic work-sharing model allows dynamic addition of tasks, as the DAG is being processed. While the static work-sharing model facilitates global scheduling decisions, the dynamic model provides

greater flexibility.

In this paper, we propose a programming interface (API) for the user to specify the DAG to the run-time system. We then discuss issues in developing an efficient implementation of the API.

This extended abstract is organized as follows. Computations and kernels representative of the load-balancing design pattern are presented in Section 2. Section 3 briefly describes the Global Arrays programming model. The proposed API for the work-sharing construct is discussed in Section 4. In Section 5, we discuss implementation issues.

2 Motivating Examples

2.1 Tensor Contraction Expressions

The Tensor Contraction Engine (TCE) [1, 3] 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 the product of several block-sparse input arrays. The dimensions of all the arrays are divided into certain number of segments and these segments are ordered. An operation on the indices of the segments that form a block of an array are used to determine if it is non-zero. Each non-zero block in an input or intermediate matrix may be involved in more than one matrix multiply between blocks. The sequence of tensor contraction expressions with block-sparsity can be represented as a DAG structure. The wide-ranging sizes of the blocks leads to significant variation in 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.

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 [12]. Symmetry of forces between two particles halves the amount of computation. Thus it is sufficient to compute the lower triangular force matrix and block decomposition of lower triangular matrix leads to load imbalance. A centralized task list (work queue of blocks, where number of blocks is greater than the number of processors by at least a factor of 2) stores the information about the order in which forces are computed. Initially, a global atomic counter is set to zero. The global counter is an atomic variable, i.e. it has mutual exclusion built in and concurrent accesses are serialized. It corresponds to the index of the next available task (i.e block info) in the task list. All the processes get the task index from the global counter, update the global counter and get the corresponding block from the task list. When a process finishes computing its block, it gets the next available task from the task list.

2.3 Parallel Dense Matrix Multiplication

In many scientific applications, the matrix distribution is based on the underlying physical problem and might involve variable block sizes on individual processors, leading to load imbalance. Therefore, each matrix can be logically partitioned into equal-sized blocks, independent of the underlying distribution. The assignment of the logical blocks of matrices to individual processors can be determined at run-time to achieve load balancing. A local task list is built that contains information of the logical blocks to be computed. Once the physical location of the logical block in the distributed/partitioned address space is determined, indices of the corresponding logical block need to be determined. With this information available, the appropriate communication calls transfer the relevant data. When a process finishes computing an update of the logical block in the result matrix, it gets the next logical block to be computed from the task list.

3 Global Array Programming Suite

The Global Arrays programming suite 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

```

work_pool wp

work_element we {
    fn_handle,          /*function to execute*/
    input_wp_list,     /*input work elements*/
    input_ga_list,     /*input global arrays*/
    input_ga_range_list, /*ranges for input global arrays*/
    result_dims,       /*output array dimensions*/
    output_ga,         /*output global array*/
    output_ga_range    /*ranges for output global arrays*/
}

wp = create_work_pool()

add_work_element(wp, we)

add_work_element_list(wp, we_list)

seal_work_pool(wp)

process_work_pool(wp)

```

Figure 1: API to specify the computation DAG

communication. The Aggregate Remote Memory Copy Interface (ARMCI) library [9] provides a distributed-memory view with remote memory access. 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 [2], GPShMEM [11], and Global Arrays.

The next higher level is the Global Arrays (GA) library [10] [6]. GA 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 [4] (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 [8].

The Disk Resident Arrays (DRA) model [5] 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 provide high performance while providing a programming model that is simpler than message passing.

4 Computation Specification

A set of routines, shown in Fig. 1 is provided to the user to specify the DAG. For each DAG, all processes collectively create a *work_pool* object (*wp*), using the *create_work_pool* method.

The actual representation of the work pool object is determined by the language binding. For example, this could be an integer handle in C and Fortran, or an object in C++. A *work_element* object (*we*) encodes each task. It specifies the function to be executed to process the work element, the list of work elements it depends on and global arrays it takes as input, the size of the output produced and the output global array to contain the result.

The function handle is a globally unique identifier for the method that processes the work element. The data

produced by input work elements are completely used. If the work element is a “sink” in the DAG, the result is written to the global array specified by *output_ga*, in the region given by the *output_ga_range* field. The memory to store the results of the non-sink work elements is automatically by the system using the sizes specified using the *result_dims* field. This dynamic memory allocation obviates any communication that might result from an advance binding of the memory, by the user, to a processor.

The global array handles might be replaced by ARMCI or DRA object handles, depending on the data location. In general, any memory region, accessible in a one-sided fashion, can be used.

The created work elements can be added to a work pool using the *add_work_element* and *add_work_element_list* methods. The creation and addition of work elements to the work pool is done by one of the processes, after the collective creation of the work pool.

In the static work-sharing model, the *seal_work_pool* method can be used to *seal* the work pool. This notifies the runtime system that no more additions will be made, allowing effective start-time optimizations.

Subsequently, all the processes collectively invoke *process_work_pool* to process the work elements in the work pool. Fig. 2 illustrates block-sparse matrix multiply with the work-sharing API, in the C language. The multiplication is of the form

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

All dimensions are assumed to be divided into *nblocks* segments. The binary operator *op* is applied to the segment indices to determine if a block is non-zero. Parameters *g_a*, *g_b*, and *g_c* correspond to *A*, *B*, and *C* arrays, respectively, while *rank* is the rank of the process. The matrix multiplication is divided into two parts: computation of the partial products and addition of the partial products. Methods *get_ranges* and *get_size* are used to compute the ranges and sizes of the non-zero block with the given segment indices.

5 Efficient Work-sharing Implementation

Several research issues must be addressed to obtain an efficient implementation of the work-sharing construct. The Global Arrays programming suite needs to be extended to support additional data structures and access mechanisms.

In addition, algorithms need to be developed to schedule the computation corresponding to the worksharing construct to achieve locality-aware load-balancing and minimize the overall execution time. A load-balanced processing of the work pool requires global information about completed and remaining work elements. This is especially important for the dynamic work-sharing model. We are considering a global data structure, such as a global work queue, for that purpose.

The Global Arrays programming suite, in its latest version, distributes data amongst all the processes in a process group [7]. Creation of process groups is a collective operation. An asynchronous model of load-balancing would require disjoint subsets of processes to operate independent of each other. On the other hand, knowledge of the location and distribution of data is essential to process subsequent work elements. A solution would be to allow un-coordinated allocation of array in memory while processing its producer, while deallocating it when the last consumer has been processed. Any memory region or global array created for intermediate results are known only to the run-time system and are invisible to the user.

The amount of memory required for the computation can exceed the available physical memory. Each task should independently be able to write output data to and read input data from the disk. Hence, we have recently added one-sided (noncollective) I/O to the DRA model. The DAG can potentially be divided into sub-DAGs, each of which can be processed without I/O. Thus all required on disk can be read into memory, the sub-DAG processed, and the results written back to disk. This mechanism might not be sufficient for the dynamic work-sharing model, in which the memory requirement is not at start-time. In this case, a transparent mechanism, similar to virtual memory, to move data to/from disk can be supported. The tasks to be executed are ordered based on priority and data required by high priority tasks can be kept in memory at the expense at lower priority tasks. One priority mechanism would be the expected time to start processing a task, with the task expected to be processed in the immediate future having a higher priority.

Optimizing the execution involves determining schedules that optimize the cumulative cost, including computation, communication, and disk I/O cost, while ensuring that the memory utilized at any point during the computation does not exceed the available memory. Algorithms that determine schedules based on the various costs are to be explored. In addition, heuristics to select, at runtime, the tasks to process, given partial information on the state of processing of the DAG, are needed.

```

void matmul(g_c, g_a, g_b, nblocks, rank) {
    work_pool wp = create_work_pool();
    work_element *we, *wel;

    if(rank == 0) {
        for(i=0; i<nblocks; i++) {
            for(j=0; j<nblocks; j++) {
                if(op(i,j) == 0) /*zero C block*/
                    continue;

                we = malloc(sizeof(work_element));
                we->fn_handle = SUM_HANDLE;
                we->output_ga = g_c;
                we->output_range = get_ranges(i, j);
                add_work_element(wp, we);

                for(k=0; k<nblocks; k++) {
                    if(op(i,k)==0 or op(k,j)==0) /*zero A or B block*/
                        continue;

                    wel = malloc(sizeof(work_element));
                    wel->fn_handle = DGEMM_HANDLE;
                    list_add(wel->input_ga_list, g_a);
                    list_add(wel->input_ga_list, g_b);
                    list_add(wel->input_ga_range_list, get_ranges(i,k));
                    list_add(wel->input_ga_range_list, get_ranges(k,j));
                    wel->result_dims = get_size(i, j);
                    add_work_element(wp, wel);

                    list_add(we->input_wp_list, wel);
                }
            }
        }
        seal_work_pool(wp);
    }
    process_work_pool(wp);
}

```

Figure 2: Block-sparse matrix multiply with the work-sharing API

Each task in the DAG might itself possess parallelism and might need to be scheduled on more than one processor. This would require more than a simple asynchronous model.

6 Conclusion

In this paper, we proposed a work-sharing construct for dynamic and irregular computations expressible as directed acyclic graphs. The proposed API was detailed and the issues to be dealt with in obtaining an efficient implementation were discussed. We plan to include the construct as part of the Global Arrays programming suite. This feature should simplify development of dynamic load balancing in GA application codes while providing numerous opportunities for efficient implementation in the run-time layer.

References

- [1] 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. Supercomputing 2002*, November 2002.
- [2] Cristian Coarfa, Yuri Dotsenko, and John Mellor-Crummey. A Multi-Platform Co-Array Fortran Compiler. In *Proc. of the 13th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2004.
- [3] D. Cociorva, J. Wilkins, G. Baumgartner, P. Sadayappan, J. Ramanujam, M. Nooijen, D. Bernholdt, and R. Harrison. Towards Automatic Synthesis of High-Performance Codes for Electronic Structure Calculations: Data Locality Optimization. In *Proc. Intl. Conf. on High Performance Computing*, volume 2228, pages 237–248. Springer-Verlag, 2001.
- [4] High Performance Computational Chemistry Group. *NWChem, A Computational Chemistry Package for Parallel Computers, Version 4.6*. Pacific Northwest National Laboratory, Richland, Washington 99352–0999, USA, 2004.
- [5] 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.
- [6] J. Nieplocha, R. Harrison, and R. Littlefield. Global Arrays: A Nonuniform Memory Access Programming Model for High-Performance Computers. *The Journal of Supercomputing*, 10:197–220, 1996.
- [7] J. Nieplocha, M. Krishnan, B. Palmer, V. Tipparaju, and Y. Zhang. Extending scalability of the global arrays shared memory programming model using processor groups. In *Proc. of ACM Computing Frontiers*. Shermann Publishing, 2005.
- [8] 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.
- [9] Jarek Nieplocha and Bryan 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.
- [10] Jarek Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global arrays: a portable programming model for distributed memory computers. In *Supercomputing*, pages 340–349, 1994.
- [11] 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.
- [12] 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.