WorkQ: A Many-Core Producer/Consumer Execution Model Applied to PGAS Computations

David Ozog, Allen Malony
Dept. of Computer and Information Science
University of Oregon
Eugene, Oregon 97403
Email: {ozog, malony}@uoregon.edu

Jeff R. Hammond
Parallel Computing Lab
Intel Corporation
Hillsboro, Oregon 97124
Email: jeff_hammond@acm.org

Pavan Balaji
Math. and Computer Science Div.
Argonne National Laboratory
Lemont, Illinois 60439
Email: balaji@anl.gov

***Abstract*—Partitioned global address space (PGAS) applications, such as the Tensor Contraction Engine (TCE) in NWChem, often apply a one-process-per-core mapping in which each process iterates through the following work-processing cycle: (1) determine a work-item dynamically, (2) get data via one-sided operations on remote blocks, (3) perform computation on the data locally, (4) put (or accumulate) resultant data into an appropriate remote location, and (5) repeat the cycle. However, this simple flow of execution does not effectively hide communication latency costs despite the opportunities for making asynchronous progress. Utilizing nonblocking communication calls is not sufficient unless care is taken to efficiently manage a responsive queue of outstanding communication requests. This paper presents a new runtime model and its library implementation for managing tunable "work queues" in PGAS applications. Our runtime execution model, called WorkQ, assigns some number of on-node "producer" processes to primarily do communication (steps 1, 2, 4, and 5) and the other "consumer" processes to do computation (step 3); but processes can switch roles dynamically for the sake of performance. Load balance, synchronization, and overlap of communication and computation are facilitated by a tunable nodewise FIFO message queue protocol. Our WorkQ library implementation enables an MPI+X hybrid programming model where the X comprises SysV message queues and the user's choice of SysV, POSIX, and MPI shared memory. We develop a simplified software mini-application that mimics the performance behavior of the TCE at arbitrary scale, and we show that the WorkQ engine outperforms the original model by about a factor of 2. We also show performance improvement in the TCE coupled cluster module of NWChem.**

***Keywords*—Producer/Consumer, PGAS, Global Arrays, Tensor Contractions, Quantum Chemistry, Performance Evaluation**

I. Introduction

Many distributed-memory computational applications undergo a basic flow of execution: individual processes determine a task to complete, receive data from remote locations, compute on that data, send the result, and repeat until all tasks are handled or convergence is reached. Accomplishing this in a performance-optimal manner is complicated by communication wait times and variability of the computational costs among difference tasks. Several execution models and programming models strive to account for this by supporting standard optimization techniques, yet certain conditions exist for doing so effectively. In message-passing applications, for example, nonblocking communication routines must be preferred over blocking routines in order to hide the latency cost of communication. However, processes must also be capable of making *asynchronous progress* while communication occurs. At the same time, the balance of workload across processor cores must be maintained so as to avoid starvation and synchronization costs. If the variability of task execution time is not considered when incorporating latency-hiding optimizations, then suboptimal performance occurs. For instance, if computation of a task finishes before a previous nonblocking routine completes, then starvation occurs despite the asynchronous progress. In order to eliminate this problem in such applications, there must exist a dynamic queue of task data in shared memory that accommodates irregular workloads. However, if this queue becomes overloaded with data relative to other queues on other compute nodes, then load imbalance and starvation also occur despite asynchrony. This paper introduces and analyzes an execution model that accomplishes zero wait-time for irregular workloads while maintaining systemwide load balance.

Irregularity within computational applications commonly arises from the inherent sparsity of real-world problems. Load imbalance is a result of data decomposition schemes that do not account for variation due to sparsity. Not only is there fundamental variation in task dimensions associated with work items from irregular sparse data structures, but the variety and nonuniformity of compute node architectures and network topologies in modern supercomputers complicate the wait patterns of processing work items in parallel. For example, a process that is assigned a work item may need to wait for data to be migrated from the memory space of another process before computation can take place. Not only does incoming data often cross the entire memory hierarchy of a compute node; it may also cross a series of network hops from a remote location. The contention on these shared resources in turn complicates the development of distributed computational algorithms that effectively overlap communication and computation while efficiently utilizing system resources.

While nonblocking communication routines enable asynchronous progress to occur within a process or thread of execution, care must be taken to minimize overheads associated with overlapping. Polling for state and migrating data too often between process spaces can be expensive. Also, often some local computation must occur before communication can take place. The Tensor Contraction Engine (TCE) of NWChem, the target application of this paper, exhibits this behavior because a relatively sizable amount of local computation takes place to determine the global location of sparse tensor blocks before

communication can take place. For these reasons, performance may be best when the communication sections have a dedicated core, especially in modern many-core environments, where sacrificing some cores for communication may result in the most optimal mapping for latency hiding.

In this paper we study a new execution model, called WorkQ, that prioritizes the overlap of communication and computation while simultaneously providing a set of runtime parameters for architecture-specific tuning. This model achieves effective load balance while eliminating core starvation due to communication latency. Using an implementation of the WorkQ model, we perform various experiments on a benchmark that mimics the bottleneck computation within an important quantum many-body simulation application (NWChem), and we show good performance improvement using WorkQ. Section II provides the necessary background regarding the partitioned global address space (PGAS) model and the NWChem application. Section III discusses the motivation for constructing our execution model. Section IV outlines the design and implementation of WorkQ, Section V describes a set of experimental evaluations, Section VI discusses related work, and Section VII summarizes our conclusions and briefly discusses future work.

II. Background

In this section, we provide the necessary background information for understanding the context of the WorkQ model and its applications presented in this paper. Topics include *A)* the PGAS paradigm, *B)* the Global Arrays programming model, *C)* NWChem and the coupled cluster technique, and *D)* the Tensor Contraction Engine.

A. PGAS

The availability and low cost of commodity hardware components have shaped the evolution of supercomputer design toward distributed-memory architectures. While distributed commodity-based systems have been a boon for effectively and inexpensively scaling computational applications, they have also made it more difficult for programmers to write efficient parallel programs. This difficulty takes many forms: handling the diversity of architectures, managing load balance, writing scalable parallel algorithms, exploiting data locality of reference, and utilizing asynchronous control, to name a few. A popular parallel programming model that eases the burden on distributed-memory programmers is found in PGAS languages and interfaces.

In the PGAS paradigm, programs are written single program, multiple data (SPMD) style to compute on an abstract global address space. Abstractions are presented such that global data can be manipulated as though it were located in shared memory, when in fact data is logically partitioned across distributed compute nodes with an arbitrary network topology. This arrangement enables productive development of distributed-memory programs that are inherently conducive to exploiting data affinity across threads or processes. Furthermore, when presented with an application programming interface (API) that exposes scalable methods for working with global address space data, computational scientists are empowered to program vast cluster resources without having to worry about optimization, bookkeeping, and portability of relatively simple distributed operations.

Popular PGAS languages/interfaces include UPC, Titanium, Coarray Fortran, Fortress, X10, Chapel, and Global Arrays. The implementation in this work was built on top of Global Arrays/ARMCI, which is the subject of the next section.

B. Global Arrays

Global Arrays (GA) is a toolkit for doing PGAS computations in high-performance computing codes using C/C++, Fortran, or Python [13]. It is built on top of the aggregate remote memory copy interface (ARMCI), which provides efficient one-sided communication primitives optimized for most remote direct memory access (RDMA) hardware [14]. To understand the utility of GA, consider the transpose operation of a matrix in global memory. Mathematically, this is a simple operation, but it can involve intensive bookkeeping to program a global transpose in distributed memory. This is a task many computational scientists would rather avoid. GA provides the means for accomplishing transposition of a global matrix with one call that is portable and optimized to efficiently utilize one-sided RDMA operations with ARMCI. Besides the standard put/get/accumulate functionality common in one-sided communication libraries, the GA API provides a number of other helpful computational operations, including functions for matrix addition/multiplication/diagonalization/inversion, ghost cell control, strided gets and puts, and solving linear systems of equations.

A common misconception is GA's relationship with the Message Passing Interface (MPI). Although a large portion of GA's communication is done strictly through ARMCI, GA still requires linking with a message-passing library. This library need not be MPI (an alternative is TCGMSG [7]), but there does need to be a message-passing library underneath the GA stack that provides SPMD capability, process IDs, synchronization, broadcast and reduction operations, and so forth. As of this writing, MPI is the de facto standard library for satisfying these requirements. In addition, one can replace the entire ARMCI communication layer with equivalent MPI 3.0 RMA routines for doing one-sided communication [3]. This approach is typically done on newer systems and interconnects to take advantage of MPI's portability.

C. NWChem and Coupled Cluster

The NWChem computational chemistry framework is a popular open-source software package designed to support the scalability of a wide variety of methods on high-performance computer systems. NWChem is known for its strong capabilities in *ab initio* quantum chemistry methods invoking molecular electron structure theory, yet it also provides modules for simulating classical molecular dynamics. Furthermore, the QM/MM module enables the exploration of hybrid simulations that combine quantum mechanics computations in regions of high interest (such as a protein binding site) with molecular mechanics calculations in regions of lesser interest [21].

The coupled cluster (CC) component of NWChem is a important molecular electronic structure module highly utilized by the quantum chemistry and physics communities [2]. CC is

a numerical technique for solving the electronic Schrödinger equation using an exponential ansatz operator sum acting upon a one-electron reference wave function [12]. A detailed explanation is beyond the scope of this paper, but it suffices to say that the sum of operators is truncatable to arbitrary-order accuracy (analogous to a Taylor series expansion). Each operator is evaluated via a series of tensor contractions (described in the next section). When truncating CC to include only the "doubles-order" term, the method is referred to as CCD. When including both singles and doubles, the method is called CCSD. With triples and quadruples, the methods are called CCSDT and CCSDTQ, respectively. Moreover, important perturbative methods (such as CCSD(T) and CCSD(Q)) exist that can approximate the addition of a higher-order term without requiring the full increase in computational and memory requirements.

D. Tensor Contraction Engine

The Tensor Contraction Engine is a domain-specific language for automatically generating high-performance programs which compute the working equations of second quantized many-electron theories, such as coupled cluster [8]. The primary motivation for supporting such a tool is that symbolic manipulation of these equations is an extremely time consuming and error-prone process when done by hand: the TCE facilitates the generation of portable and efficient parallel code that is verified for correctness. TCE is a core component of *ab initio* chemistry capabilities in NWChem, as well as in UTChem, developed at the University of Tokyo [22].

TCE generates GA programs written in Fortran that exploit spin, spatial, and index permutation symmetries among the working set of equations to reduce the computational and memory requirements of these methods. Despite these efforts, the computations of CC methods have polynomial algorithmic complexity in terms of the number of FLOPS and memory usage. For example, CCSD equations have algorithmic complexity of $O(n^6)$ for operations and $O(n^4)$ for memory (where n is the sum of occupied and virtual electron orbitals).

The overall TCE computation consists of several Jacobi iterations through a directed acyclic graph where each node refers to a calculation of a tensor contraction intermediate (corresponding to the truncatable sum of operators described in the previous section). Before computation begins, the GA data is arranged into tiles that each contain orbitals with the same spin and spatial symmetries. The granularity of these tiles is crucial for performance because it determines the total number of work items. The tile size must be small enough for there to be more tasks than the number of processes in the application. At the same time, the tile size must be sufficiently large because an excessive number of work items leads to unnecessary accumulation of overhead on the dynamic load balancer [17].

TCE reduces the contraction of two high-dimensional tensors into a summation of the product of several 2D arrays. Therefore, the performance of the underlying BLAS library strongly influences the overall performance of TCE. For the purposes of this paper, each tensor contraction routine can be thought of as a global task pool of tile-level 2D DGEMM (double-precision general matrix-matrix multiplication) operations. This pool of work items is processed according to the following execution model:

1) A unique work item ID is dynamically assigned via an atomic read-modify-write operation to a dynamic load balancing counter (see [17] for details).
2) The global addresses of two tiles (A and B) in the global array space is determined (TCE hash lookup).
3) The corresponding data is copied to the local process space (via one-sided RMA) with `GA_Get()` calls.
4) A contraction is formed between the local copies of tiles A and B and stored into C. When necessary, a permute-DGEMM-permute pattern is performed in order to arrange the indices of the tensor tiles to align with the format of matrix-matrix multiplication.
5) Steps 2, 3, and 4 repeat over the work-item tile bundle; then C is accumulated (`GA_acc()` call) into a separate global array at the appropriate location.

Although this algorithm is specific to CC, we note we that it falls under a more general `get/compute/put` model that is common to many computational applications. For example, the problem of numerically solving PDEs on domains distributed across memory spaces certainly falls under this category.

The next section discusses motivation for the development of an alternative execution model that is able to perform this same computation more efficiently.

III. Motivation

In this section we briefly present performance measurements that support our motivation for developing a new runtime execution model for processing tasks in applications such as the TCE-CC within Global Arrays. We begin by considering measurements from a simple trace of a tensor contraction kernel. We then discuss the implications of a new execution model.

A. Communication/Computation Overlap

In order to better understand the performance behavior of the TCE task execution model described at the end of section II-D, we develop a mini-application that executes the same processing engine without the namespace complications introduced by quantum many-body methods. The details of this mini-app will be discussed in Sections IV-B and V-A, but here we present a simple trace and profile of the execution to better motivate and influence the design of our runtime in Section IV.

The (A) and (B) portions of Fig. 1 show an excerpt of a trace collected with the TAU parallel performance system [18] for 12 MPI processes on 1 node within a 16 node application executed on the ACISS cluster (described in Section V). This trace is visualized in Jumpshot with time on the horizontal axis, each row corresponding to an MPI process and each color corresponding to a particular function call in the application. Specifically, the purple bars correspond to step 1 in the TCE execution model described in the previous section. The green bars correspond to the one-sided get operation on the two tiles A and B from step 3 (step 2 is implicit in the mini-app and is thus not contained in a function). The yellow bars are non-communication work cycles, and the pink bars are

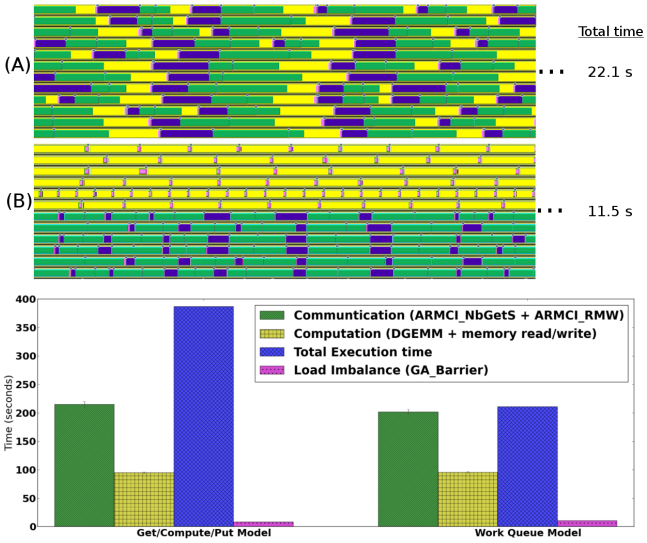

Fig. 1. (Top:) TAU trace of original code (A) compared to WorkQ (B) for a 192 MPI process job. Yellow is computation, purple is `ARMCI_Rmw` (corresponding to `Nxtval` call), green is `ARMCI_NbGetS` (corresponding to `GA_Get` call), and pink is `DGEMM`. (Bottom:) Profile information comparing the two execution models for a larger 960 MPI process job. The work queue implementation accomplishes full overlap without sacrificing load balance.

DGEMM (these are small because of the relatively small tile size in this experiment). Yellow and pink together correspond to step 4. Step 5 is not shown in this timeline but occurs at a future point at the end of this task bundle. The 12 rows in portion (B) show a corresponding trace with our alternative parallel execution model in which 6 MPI processes dedicate their cycles to computation, and the other 6 processes dedicate their cycles to communication.

The bottom half of Fig. 1 contains timing information extracted from TAU profiles for a larger job with 960 MPI processes. The measurements clearly show that the work queue execution model accomplishes effective overlap of communication with computation without sacrificing load imbalance. This results in a speedup of almost 2x over the original get/compute/put model for this experiment. The advantage can be inferred from the trace: in the original execution, there are moments when hardware resources are fully saturated with computation (i.e., all rows are yellow at a particular time) yet other moments where starvation is occurring (i.e., rows are green at a particular time). Besides dramatically reducing moments of work starvation, the alternative model enables tunability: for instance, the optimal number of computation versus communication processes can be determined empirically.

B. Variability in Work-Item Processing

The TCE engine uses blocking `GA_get()` and `GA_acc()` calls to gather and accumulate tiles, respectively, into the global space. While it is reasonable to use the corresponding nonblocking API for GA (`GA_nbget` and `GA_nbacc`) to accomplish overlap, doing so will not achieve optimal performance in the face of highly irregular workloads. For example, one can submit a nonblocking get before doing computation on local data; but if the computation finishes before the communication request is satisfied, then starvation occurs. Variation in execution time occurs often, because of either system noise or inherent differences in task sizes. This variablity necessitates the calling of multiple nonblocking communication operations managed by a queue so that data is always available once an iteration of computation finishes. On the other hand, the number of work items in this queue must be throttled so that the queue does not become overloaded with respect to other queues on other nodes. If this were to happen, then load imbalance would surely occur without the usage of techniques such as internode work stealing, which potentially incur high overheads.

IV. Design and Implementation

The desire to overlap communication and computation in a dynamic and responsive manner motivates the development of a library for managing compute node task queuing and processing within SPMD applications. We have implemented such a library, which we call WorkQ. This section presents the software architectural design for WorkQ, describes some implementation details and possible extensions, and presents a portion of the API and how it can be deployed for efficient task processing in distributed memory SPMD programs.

A. WorkQ Library

As described in Section III, the TCE-CC task-processing engine has the potential to experience unnecessary wait times and relatively inefficient utilization of local hardware resources. Here we describe an alternative runtime execution model with the goals of (1) processing tasks with less wait time and core starvation, (2) exposing tunability to better utilize hardware resources, and (3) responding dynamically to real-time processing variation across the cluster.

Here we simplify the operations of TCE described in Section II-D into a pedagogical model that is akin to tiled matrix multiplication of two arrays, A and B . In this model, A and B contain GA data that is distributed across a cluster of compute nodes. The overall goal of the application is to multiply corresponding tiles of A and B, then to accumulate the results into the appropriate location of a separate global array, C. In order to accomplish this within the original execution engine, individual processes each take on the execution loop from Section II-D: `get(A); get(B); compute(A,B); acc(C)`. The behavior of a single compute node involved in this computation is characterized by the trace in the top half of Fig. 1: at any given moment in time, processes work within a particular stage of the execution loop.

In the WorkQ runtime, each compute node contains an FIFO message queue, $\mathbf{Q}_1$, in which some number of *courier processes* are responsible for placing A and B tile metadata onto $\mathbf{Q}_1$, then storing the incoming tiles into node-local shared memory. Meanwhile, the remaining *worker processes* dequeue task metadata bundles as they become available, then use this information to follow a pointer to the data in shared memory and perform the necessary processing. Once a worker process is finished computing its task result, it places the resultant data into a separate FIFO message queue, $\mathbf{Q}_2$, which contains data for some courier process to eventually accumulate into C.

We now describe the four primary components of the WorkQ implementation: the dynamic producer/consumer sys-

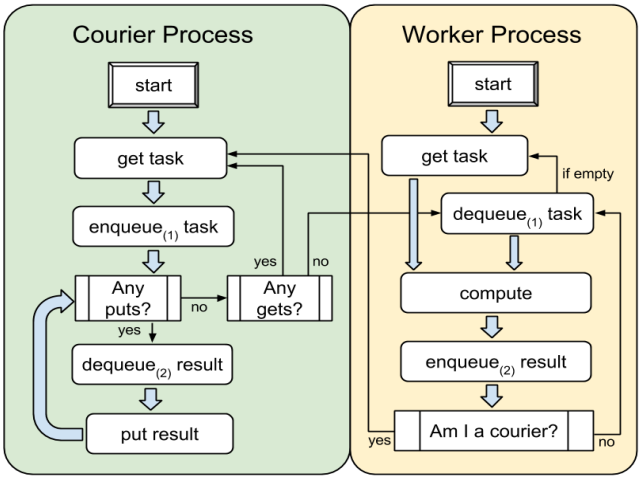

Fig. 2. Flow diagram for the dynamic producer/consumer version of the WorkQ execution model. The left green column represents the activities of courier processes, and the right yellow column represents activities of worker processes. In this system, couriers can temporarily become workers and vice versa.

tem, the on-node message queues, the on-node shared memory, and the WorkQ library API.

1) Dynamic Producer/Consumer: This runtime system exhibits a form of the producer/consumer model in which courier processes are the producers and worker processes are the consumers. In the model described so far, couriers perform mostly remote communication, and workers constantly read-/write data from/to local memory and perform a majority of the FLOPS in this phase of the application. However, we found via performance measurements that this system can still struggle with unacceptable wait times and starvation from the point of view of the workers. This situation occurs, for example, when $\mathbf{Q}_1$ is empty because of either relatively long communication latencies or not enough couriers to keep the workers busy. For this reason, the WorkQ implementation allows for the dynamic switching of roles between courier and worker.

Figure 2 displays how role switching occurs in the WorkQ runtime. To bootstrap the system, *both* couriers and workers perform an initial `get()` operation. This "initialization" of $\mathbf{Q}_1$ is done to avoid unnecessary work starvation due to an empty initial queue. We determined this to be critical for performance, especially when the time to get/enqueue a task is greater than or equal to the compute time. If this is the case (and the number of workers approximately equals the number of couriers), the workers may experience several rounds of starvation before the couriers can "catch up."

After the first round (with workers computing on tiles they themselves collected), workers dequeue subsequent tasks to get data placed in $\mathbf{Q}_1$ by the couriers. If $\mathbf{Q}_1$ ever becomes empty when a worker is ready for a task, the worker will get its own data and carry on. On the other hand, if a courier finds either that $\mathbf{Q}_1$ is overloaded (as determined by a tunable runtime threshold parameter) or that $\mathbf{Q}_2$ is empty with no remaining global tasks, then the courier will become a worker, dequeue a task, and compute the result. In either case, the process will return to its original role as a courier until both $\mathbf{Q}_1$ and $\mathbf{Q}_2$ are empty.

2) Message Queues: The nodewise metadata queues are implemented by using the System V (SysV) UNIX interface for message queues. This design decision was made because SysV message queues exhibit the best trade-off between latency/bandwidth measurements and portability compared with other Linux variants [19]. Besides providing atomic access to the queue for both readers and writers, SysV queues provide priority, so that messages can be directed to specific consumer processes. For example, this functionality is utilized to efficiently end a round of tasks from a pool: when a courier is aware it has enqueued the final task from the pool, it then enqueues a collection of finalization messages with a process-unique `mtype` value corresponding to the other on-node process IDs.

3) Shared Memory: The message queues just described contain only metadata regarding tasks; the data itself is stored elsewhere in node-local shared memory. This approach is taken for three reasons: (1) it reduces the cost of contention on the queue among other node-resident processes, (2) Linux kernels typically place more rigid system limits on message sizes in queues (as seen with `ipcs -l` on the command line), and (3) the size and dimension of work items vary significantly. The message queue protocol benefits in terms of simplicity and performance if each queued item has the same size and structure. Within each enqueued metadata structure are elements describing the size and location of the corresponding task data. The WorkQ library allows for either SysV or POSIX shared memory depending on user preference. There is also an option to utilize MPI_3 shared-memory windows (`MPI_Win_allocate_shared`) within a compute node. This provides a proof of concept for doing MPI+MPI [9] hybrid programming within the WorkQ model.

4) Library API: The WorkQ API provides a productive and portable way for an SPMD application to initialize message queues on each compute node in a distributed-memory system, populate them with data, and dequeue work items. Here we list a typical series of calls to the API (because of space constraints, arguments are not included; they can be found in the source [16]):

- `workq_create_queue()`: a collective operation that includes on-node MPI multicasts of queue info
- `workq_alloc_task()`: pass task dimensions and initialize pointer to user-defined metadata structure
- `workq_append_task()`: push a microtask's data/metadata onto the two serialized bundles
- `workq_enqueue()`: place macrotask bundle into the queue then write real-data into shared memory
- `workq_dequeue()`: remove a macrotask bundle from the queue and read data from shared memory
- `workq_get_next()`: pop a microtask's metadata and real data in preparation for computation
- `workq_execute_task()`: (optional) a callback so data can be computed upon with zero copies
- `workq_free_shm()`: clean up the shared memory
- `workq_destroy()`: clean up the message queues

WorkQ also includes a wrapper to SysV semaphores, which is needed only if the explicit synchronization control is needed (i.e., if certain operations should not occur while workers are computing). These functions are `workq_sem_init()`, `sem_post()`, `sem_release()`, `sem_getvalue()`, and `sem_wait()`.

B. TCE Mini-App

The performance of the WorkQ runtime system implementation is evaluated in two ways: directly, with the original NWChem application applied to relevant TCE-CC ground-state energy problems, and indirectly, with a simplified mini-app that captures the overall behavior of the TCE performance bottleneck (described in Section II-D). The primary advantage of the mini-app is that it removes the need to filter through the plethora of auxiliary TCE functionalities, such as the TCE hash table lookups, or the many other helper functions within TCE. Although the mini-app will not compute any meaningful computational chemistry results, it captures the performance behavior of TCE in a way that is more straightforward to understand and simpler to tune. Furthermore, the tuned runtime configuration within the mini-app environment can be applied to NWChem on particular system architectures.

The TCE mini-app implements the pedagogical model described in Section IV-A: corresponding tiles from two global arrays (A and B) are multiplied via a DGEMM operation and put into a third global array C. The mini-app is strictly a weak-scaling application that allows for a configurable local buffer length allocation on each MPI process. These buffers are filled with arbitrary data in the creation/initialization of A, B, and C. As in TCE, all global arrays are reduced to their one-dimensional representations [8]. The heap and stack sizes fed to the global arrays memory allocator [13] are set to as large as possible on a given architecture. Two versions of the code are implemented to calculate the entire pool of DGEMMs: one with the original `get/compute/put` model on every process and one with the WorkQ model on every compute node. The resulting calculation is verified in terms of the final vector norm calculated on C.

V. Experimental Results

The performance of the WorkQ execution runtime compared wwitho the standard `get/compute/put` model is evaluated on two different platforms. The first is the ACISS cluster located at the University of Oregon. Experiments are run on the 128 generic compute nodes, each an HP ProLiant SL390 G7 with 12 processor cores per node (2x Intel X5650 2.67 GHz 6-core CPUs) and 72 GB of memory per node. This is a NUMA architecture with one memory controller per processor. ACISS employs a 10 gigabit Ethernet interconnect based on a 1-1 nonblocking Voltaire 8500 10 GigE switch that connects all compute nodes and storage fabric. The operating system is RedHat Enterprise Linux 6.2, and MPICH 3.1 is used with the `-O3` optimization flag.

The second platform is the Laboratory Computing Resource Center "Blues" system at Argonne National Laboratory. The 310 available compute nodes each have 16 cores (2x Sandy Bridge 2.6 GHz Pentium Xeon with hyperthreading disabled) and 64 GB of memory per node. All nodes are interconnected by InfiniBand Qlogic QDR. The operating system is Linux running kernel version 2.6.32. MVAPICH2 1.9 built with the Intel 13.1 compiler was used for our experiments.

Unless otherwise specified, performance experiments are executed with 1 MPI process per core, leaving 1 core open on each compute node for the ARMCI helper thread (for example, 11 processes per node on ACISS and 15 processes per node on Blues). Previous work has shown this mapping to be optimal for reducing execution time as suggested by detailed TAU measurements in NWChem TCE-CC [6].

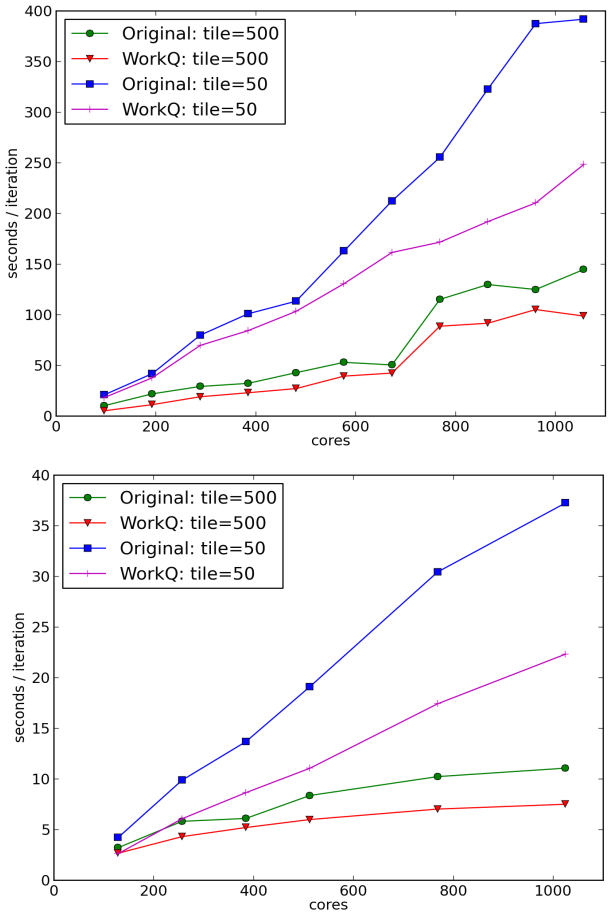

Fig. 3. Weak-scaling performance of the TCE mini-app with ARMCI over sockets on ACISS (top) and ARMCI over InfiniBand on Blues (bottom) for different tile sizes. On ACISS the WorkQ implementation was run with 6 courier processes and 5 worker processes, and on Blues with 3 couriers and 4 workers. On both architectures, the WorkQ execution shows better relative speedup with small tile sizes but better absolute performance for relatively larger tile sizes.

The systems above provide a juxtaposition of the performance benefits gained with the WorkQ runtime between two very different network interconnects: Ethernet and InfiniBand (IB). The GA/ARMCI and MPI layers utilize socket-based connections on ACISS, meaning that the servicing of message requests involves an active role of each compute node's operating system. Blues, on the other hand, has full RDMA support, so data can be transferred between nodes without involving the sender and receiver CPUs.

A. TCE Mini-App

Our first experiment considers the weak scaling performance of the TCE mini-app on ACISS and Blues for two different tile sizes. The tile size in the mini-app corresponds to the common dimension of the blocks of data collected from the GAs described in Section IV-B. In this experiment, all DGEMM operations are performed on matrices with square dimensions, $N \times N$, where N is the so-called tile size. Figure 3

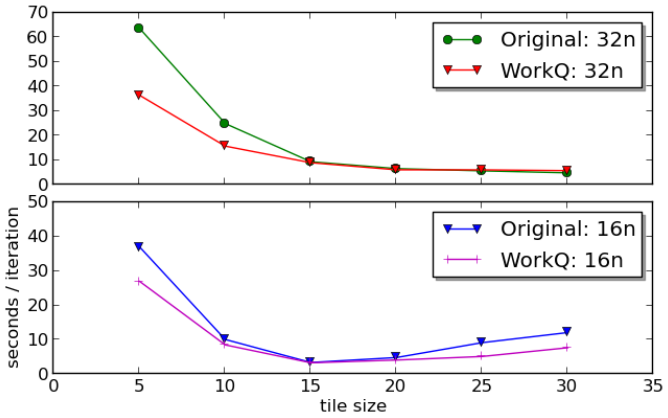

Fig. 4. Time per CCSD iteration for w3 aug-cc-pVDZ on ACISS versus tile size. The top row contains execution measurements on 32 nodes (384 MPI processes), and the bottom row contains measurements on 16 nodes (192 MPI processes).

considers tile sizes 50 (2,500 total double-precision floating-point elements) and 500 (250,000 elements). The mini-app is a weak-scaling application in which a constant amount of memory is allocated to each process/core at any given scale. That is, if the scale is doubled, then the size of the overall computation is doubled (hence, execution time increases for higher numbers of processes). GA's internal memory allocator is initialized so that the total heap and stack space per node is about 20 GB.

Figure 3 clearly shows that using the relatively large tile size of 500 results in better overall absolute performance for both the WorkQ execution model and the original execution model. This phenomenon is well understood [17] and is due mainly to the overhead associated with data management and dynamic load balancing when tile size is relatively small. In general, larger tile sizes are desirable in order to minimize this overhead, but at a certain point large tiles are detrimental to performance because it leads to work starvation. For instance, if more processes/cores are available to the application than there are number of tiles, then work starvation will surely occur.

On the other hand, the best speedups achieved with the WorkQ model on both systems are seen with the smaller tile size of 50, particularly at relatively large scales. Our TAU profiles show that at a tile size of 50, the total time spent in communication calls (`ARMCI_NbGetS` and `ARMCI_NbAccS`) is considerably larger than with a tile size of 500. These results suggest that at smaller tile sizes, there is more cumulative overhead from performing one-sided operations and therefore more likelihood that processes will spend time waiting on communication. This scenario results in more opportunity for overlap but worse absolute performance because of the incurred overhead of dealing with more tasks than necessary.

B. NWChem

We now analyze the performance of the WorkQ model applied to TCE in NWChem by measuring the time of execution to calculate the total energy of water molecule clusters. These problems are important because of their prevalence in diverse chemical and biological environments [1]. We examine the performance of the tensor contraction that consistently consumes the most execution time in the TCE CCSD calculation, corresponding to the term

$$r^{p_3 p_4}_{h_1 h_2} \mathrel{+}= \frac{1}{2} t^{p_5 p_6}_{h_1 h_2} v^{p_3 p_4}_{p_5 p_6}$$

(see [8] for details regarding the notation). In TCE, this calculation is encapsulated within routine `ccsd_t2_8()` and occurs once per iteration of the Jacobi method.

Figure 4 shows the minimum measured time spent in an iteration of `ccsd_t2_8()` on a 3-water molecule cluster using the aug-cc-pVDZ basis set across a range of tile sizes. These measurements are on the ACISS cluster at two different scales: 32 compute nodes in the top plot and 16 compute nodes in the bottom plot, with 12 cores per node in each case. Here we use the minimum measured execution time for a series of runs because it is more reproducible than the average time [5]. On 16 nodes, we see overall performance improvement with WorkQ across most measured tile sizes. As in the TCE mini-app (Fig. 3), WorkQ shows better performance *improvement* at small tile sizes but best *absolute* performance with a medium-sized tile. The performance with the small tile size is important because NWChem users do not know *a priori* which tile size is appropriate for a given problem. It is typically best to initially choose a relatively small tile size because load imbalance effects can be avoided with a finer granularity of task sizes.

VI. Related Work

Other execution and programming models incorporate node-local message queues for hiding latency and supporting the migration of work units. For example, Charm++ provides internal scheduling queues that can be used for peeking ahead and prefetching work units (called *chares*) for overlapping disk I/O with computation [10]. The Adaptive MPI (AMPI) virtualization layer can represent MPI processes as user-level threads that may be migrated like chares, enabling MPI-like programming on top of the Charm++ runtime.

Another interesting new execution model targeted to exascale development is ParalleX, which is implemented by the HPX runtime [20]. The ParalleX model extends PGAS by permitting migration of objects across compute nodes without requiring transposition of corresponding virtual names. The HPX thread manager implements a work-queue-based execution model, where parcels containing active messages are shipped between "localities." Like ParalleX, WorkQ provides the benefit of implicit overlap and load balance with the added feature of dynamic process role switching, which keeps the queue populated if too few items are enqueued and throttled if too many are enqueued. Unlike HPX and Charm++, the WorkQ library API enables such implicit performance control on top of other portable parallel runtimes, such as MPI itself and Global Arrays/ARMCI.

In execution models based on node-local message queues, work stealing enables more adaptive control over load balance. The work-stealing technique is well studied, especially in computational chemistry applications [4], [15]. In a typical work-stealing implementation, local work items are designated as tasks that may be stolen by other processes or threads. In some sense, most tasks in WorkQ are stolen from on-node

couriers by workers. Couriers work on their own tasks only if the queue is deemed overloaded, and workers take tasks from the global pool if the queue is running empty.

Novel developments in wait-free and lock-free queuing algorithms with multiple enqueuers and dequeuers [11] could potentially improve performance of this execution system by reducing contention in shared memory. SysV and POSIX queues provide atomicity and synchronization in a portable manner, but neither is wait-free or lock-free.

VII. Conclusion

The `get/compute/put` model is a common approach for processing a global pool of tasks, particularly in PGAS applications. This model suffers from unnecessary wait times on communication and data migration that could potentially be overlapped with computation and node-level activities. The WorkQ model introduces an SPMD-style programming technique in which nodewise message queues are initialized on each compute node. A configurable number of courier processes dedicate their efforts to communication and to populating the queue with data. The remaining worker processes dequeue and compute tasks. We show that a mini-application that emulates the performance bottleneck of the TCE achieves performance speedups up to 2x with a WorkQ library implementation. We also show that WorkQ improves the performance of NWChem TCE-CCSD across many tile sizes on the ACISS cluster.

Future work will include the incorporation of internode work stealing between queues and performance analysis of the queuing system using event-based simulation techniques.

Acknowledgments

D. Ozog is supported by the Department of Energy Computational Science Graduate Fellowship (DOE CSGF) program under contract DE-FG02-97ER25308. This research used resources of the Argonne Leadership Computing Facility and the Laboratory Computing Resource Center at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. The research at the University of Oregon was supported by grants from the U.S. Department of Energy, Office of Science, under contracts DE-FG02-07ER25826, DE-SC0001777, and DE-FG02-09ER25873.

References

[1] E. Aprà, A.P. Rendell, R.J. Harrison, V. Tipparaju, W.A. deJong, and S.S. Xantheas. Liquid Water: Obtaining the Right Answer for the Right Reasons. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 66:1–66:7, New York, NY, USA, 2009. ACM.

[2] R.F. Bishop. An Overview of Coupled Cluster Theory and Its Applications in Physics. *Theoretica chimica acta*, 80(2-3):95–148, 1991.

[3] J. Dinan, P. Balaji, J.R. Hammond, S. Krishnamoorthy, and V. Tipparaju. Supporting the Global Arrays PGAS Model Using MPI One-Sided Communication. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 739–750, May 2012.

[4] J. Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable Work Stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 53:1–53:11, New York, 2009. ACM.

[5] William Gropp and Ewing Lusk. Reproducible Measurements of MPI Performance Characteristics. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1697 of *Lecture Notes in Computer Science*, pages 11–18. Springer Berlin Heidelberg, 1999.

[6] J.R. Hammond, S. Krishnamoorthy, S. Shende, N.A. Romero, and A.D. Malony. Performance Characterization of Global Address Space Applications: A Case Study with NWChem. *Concurrency and Computation: Practice and Experience*, 24(2):135–154, 2012.

[7] R. J. Harrison. Portable Tools and Applications for Parallel Computers. *International Journal of Quantum Chemistry*, 40(6):847–863, 1991.

[8] So Hirata. Tensor Contraction Engine: Abstraction and Automated Parallel Implementation of Configuration-Interaction, Coupled-Cluster, and Many-Body Perturbation Theories. *The Journal of Physical Chemistry A*, 107(46):9887–9897, 2003.

[9] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, W. Gropp, V. Kale, and R. Thakur. MPI + MPI: A New Hybrid Approach to Parallel Programming with MPI plus Shared Memory. *Computing*, 95(12):1121–1136, 2013.

[10] Laxmikant V. Kale and Gengbin Zheng. *Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects*, pages 265–282. John Wiley & Sons, Inc., 2009.

[11] Alex Kogan and Erez Petrank. Wait-free Queues with Multiple Enqueuers and Dequeuers. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 223–234, New York, 2011. ACM.

[12] Karol Kowalski, Jeff R Hammond, Wibe A de Jong, Peng-Dong Fan, Marat Valiev, Dunyou Wang, Niranjan Govind, and JR Reimers. *Coupled Cluster Calculations for Large Molecular and Extended Systems*. Wiley: Hoboken, NJ, 2011.

[13] J. Nieplocha, R.J. Harrison, and R.J. Littlefield. Global Arrays: A Nonuniform Memory Access Programming Model for High-Performance Computers. *The Journal of Supercomputing*, 10(2):169–189, 1996.

[14] Jarek Nieplocha and Bryan Carpenter. ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems. In *Parallel and Distributed Processing*, volume 1586 of *Lecture Notes in Computer Science*, pages 533–546. Springer Berlin Heidelberg, 1999.

[15] A. Nikodem, A. V. Matveev, T. M. Soini, and N. Rösch. *International Journal of Quantum Chemistry*, 114(12):813–822, 2014.

[16] D. Ozog. TCE mini-app source code repository. https://github.com/davidozog/NWChem-mini-app.

[17] D. Ozog, J.R. Hammond, J. Dinan, P. Balaji, S. Shende, and A. Malony. Inspector-Executor Load Balancing Algorithms for Block-Sparse Tensor Contractions. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 30–39, Oct 2013.

[18] Sameer S. Shende and Allen D. Malony. The TAU Parallel Performance System. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.

[19] W. Richard Stevens. *UNIX Network Programming, Volume 2 (2nd ed.): Interprocess Communications*. Prentice Hall PTR, Upper Saddle River, NJ, 1999.

[20] A. Tabbal, M. Anderson, M. Brodowicz, H. Kaiser, and T. Sterling. Preliminary Design Examination of the Parallex System from a Software and Hardware Perspective. *SIGMETRICS Perform. Eval. Rev.*, 38(4):81–87, March 2011.

[21] M. Valiev, E.J. Bylaska, N. Govind, K. Kowalski, T.P. Straatsma, H.J.J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T.L. Windus, and W.A. de Jong. NWChem: A Comprehensive and Scalable Open-Source Solution for Large Scale Molecular Simulations. *Computer Physics Communications*, 181(9):1477–1489, 2010.

[22] T. Yanai, H. Nakano, T. Nakajima, T. Tsuneda, S. Hirata, Y. Kawashima, Y. Nakao, M. Kamiya, H. Sekino, and K. Hirao. UTChem A Program for ab initio Quantum Chemistry. In *Computational Science ICCS 2003*, volume 2660, pages 84–95. Springer Berlin Heidelberg, 2003.