Global Futures: a Multithreaded Execution Model for Global Arrays-based Applications

Daniel Chavarría-Miranda
High Performance Computing
Pacific Northwest National Laboratory
email: daniel.chavarria@pnnl.gov

Sriram Krishnamoorthy
High Performance Computing
Pacific Northwest National Laboratory
email: sriram@pnnl.gov

Abhinav Vishnu
High Performance Computing
Pacific Northwest National Laboratory
email: abhinav.vishnu@pnnl.gov

***Abstract*—We present *Global Futures* (GF), an execution model extension to Global Arrays, which is based on a PGAS-compatible active message-based paradigm. We describe the design and implementation of *Global Futures* and illustrate its use in a computational chemistry application benchmark (Hartree-Fock matrix construction using the Self-Consistent Field method). Our results show how we used GF to increase the scalability of the Hartree-Fock matrix build to 6,144 cores of an Infiniband cluster. We also show how GF's multithreaded execution has comparable performance to the traditional process-based SPMD model.**

I. Introduction

Compute nodes in large-scale HPC systems are evolving towards systems with large numbers of cores per node: i.e. Hopper, the Cray XE6 system at DOE's NERSC user facility, as well as the recently deployed K computer at Japan's RIKEN center. This trend is expected to increase as newer large-scale HPC systems are deployed on the path to exascale [1].

Even though newer systems have been deploying more total memory per compute node, the increasing number of cores per compute node implies that the available memory per core is *decreasing*. This has critical implications for the large HPC application software base. Most large-scale HPC applications have been developed using the Single Program - Multiple Data (SPMD) paradigm, which typically is realized by starting up a number of sequential processes $(0 \cdots p-1)$ on individual "processors" (cores in modern systems). Explicit parallelism is typically realized *only through this mechanism* (in contrast to implicit core-level parallelism such as Instruction Level Parallelism (ILP) or SIMD vector instructions). Thus, this model can only implement explicit parallelism in a *single level*. The current generation of HPC petascale programming models such as Message Passing Interface (MPI) [2] and Global Arrays (GA) [3], as well as PGAS language-based models such as Unified Parallel C (UPC) [4] and Co-Array Fortran [5], [6] are all based on the SPMD paradigm. SPMD models may incur a high cost in data replication due to private per-process address spaces, when considering multiple processes running on the same SMP node.

Hybrid programming with MPI and OpenMP has been used as a solution for applications that require larger amounts of memory per process, yet still want to use all available cores for computation exploiting multi-level parallelism. These hybrid solutions have achieved good performance in some cases, while in other cases it has been challenging to match the corresponding pure MPI performance [7]. There are several reasons why OpenMP implementations may not be able to match the performance of an MPI implementation running on a shared memory node using the same number of core resources: less control over memory allocation and OpenMP thread placement on NUMA systems[1], as well as the need to run all OpenMP threads in a tightly synchronized manner due to the constraints and requirements of the OpenMP *execution model*: fork-join parallelism executing in a BSP-like manner.

Next-generation PGAS models such as X10 [8] and Chapel [9] have developed an alternative execution model that is based on the ability to create activities on remote nodes in a distributed memory system (asyncs in X10, tasks in Chapel). X10 and Chapel both enable the expression of these mechanisms at a higher level in the respective languages through the mapping of distributed data structures to the compute nodes where the application is running. In this manner, an X10 `async` or Chapel task can be created on the node that owns a section of an X10 or Chapel distributed array. This alternative execution model has been termed *Asynchronous PGAS* (APGAS) [10]. The typical implementation mechanism for APGAS environments on distributed memory systems has been to use process-based SPMD execution, where each process runs a multithreaded environment. The underlying runtime libraries use an active message-based [11] approach to create activities on remote nodes, which are then executed by a dedicated thread or by a user-level compute thread within the target multithreaded process. APGAS extensions have also been proposed for Co-Array Fortran [12].

a) Global Arrays: Global Arrays (GA) is a library-based PGAS programming model, developed over the past two decades [13], [14]. It is based on providing global-view access to dense arrays distributed across the memories of a distributed system. GA uses a traditional SPMD execution model and provides global-view access to the processes participating in the SPMD execution via `put()`, `get()` and other primitives that specify array slices in terms of their global index coordinates.

This research was supported by PNNL's eXtreme Scale Computing Initiative (XSCI)

[1]OpenMP applications will allocate many data structures on the NUMA memory domain closest to the master thread, while MPI processes will typically be bound to a core and allocate all memory close to it

GA's performance and scalability are on par with MPI [3] and fully interoperable with it (a GA application can mix GA & MPI calls).

Being an MPI-interoperable library-based model offers several advantages to GA including not depending on compiler technology (except for the underlying node languages: C/C++, Fortran), as well as the ability to introduce GA constructs incrementally into very large HPC applications, which can be done on a module or even a routine-by-routine basis. This is due to the fact that GA's underlying execution model is fully compatible with MPI. Library-based models potentially suffer from lack of optimization by advanced compilers, which can better understand and improve remote data accesses in language-based PGAS models (UPC, CAF, others). Language-based models have a higher cost in terms of adoption and incremental integration into existing applications due to restrictions (or lack) of interoperability with MPI and traditional node languages. Less mature language-based models also suffer a performance penalty (compared to codes written in traditional HPC models & languages) due to compiler technology that needs further refinement [15].

b) Global Futures: The principal contribution of this paper is the *Global Futures* (GF) execution model. GF can be categorized as an APGAS execution model extension to Global Arrays. *Global Futures* is library-based to maintain full compatibility with GA and MPI, as well as fully interoperable with their underlying SPMD execution environment.

Global Futures extends GA by adding a new set of API functions that enable the execution of tasks on the locations that own *arbitrary array slices* of a GA instance. GF tasks (or *futures*) are specified as user-defined functions written in the underlying node language (C/C++, Fortran). GF provides two modes of execution for futures: **active** and **passive**, as well as a number of mechanisms for detecting completion of the futures.

The rest of this paper is organized as follows: Section II presents in detail the design and implementation of *Global Futures*; Section III discusses how we use *Global Futures* to implement the two-electron contribution phase of the Self-Consistent Field (SCF) application benchmark; Section IV presents experimental results based on the SCF application benchmark; Section V discusses related work; and finally Section VI presents our conclusions and discusses future work.

II. Global Futures

We describe the design and implementation of *Global Futures* and how it uses GA's infrastructure and interfaces with it.

A. Global Arrays Execution Environment

Global Arrays uses a process-based SPMD execution environment, in which processes are able to use one-sided communication primitives to directly access remote distributed array data, without the participation or involvement of the remote process. These primitives are implemented by GA's high-performance communication runtime layer: the Aggregate Remote Memory Copy Interface (ARMCI) [16].

In many cases, these remote accesses can be directly mapped to network hardware-supported Remote Direct Memory Access (RDMA) transfers from the remote *memory* holding the data to the calling process. There are cases, such as for irregular and multidimensional strided data access patterns, in which some support is required and the the data transfer cannot be completed fully relying on RDMA transfers.

For these scenarios, GA's software architecture provides for a *communication support agent* running on each SMP node of a distributed system. This communication support agent is typically realized as a *data server thread* spawned by the master process on each SMP node. On current GA/ARMCI-supported platforms it has been sufficient to have a single thread per SMP node to respond to communication requests.

ARMCI implements a mechanism called Global Procedure Call (GPC) [17], that enables the execution of arbitrary user code upon reception of a specific type of ARMCI message by the data server thread. GPCs are implemented by executing this arbitrary user code within the context of the data server thread on the target SMP node. The concept behind executing GPCs in this form is that the user-defined GPC callback is executing in the context of the SMP node that *owns* (i.e. has direct load/store access) to target PGAS data (GA slice). This memory-domain based, rather than process-based abstraction, enables quicker response to incoming requests and reduces runtime resource utilization (eg., connections, posted buffers, etc.) by employing process-to-node rather than process-to-process interactions.

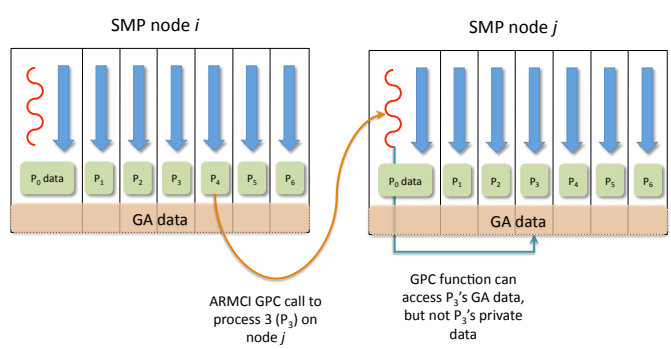

Fig. 1. ARMCI's Global Procedure Call (GPC) mechanism

Fig. 1 illustrates ARMCI's GPC mechanism: in the diagram process 4 on SMP node *i* has issued a GPC call target to process 3 on SMP node *j*. The user-defined callback for the GPC executes in the context of the data server thread owned by process 0 on SMP node *j*. The diagram indicates that all processes on SMP node *j* have direct access to GA data residing on that node, as well as process-private data.

B. Global Futures' Architecture

We extend Global Arrays' basic SPMD execution model by introducing multithreaded GA processes. This enables the

asynchronous execution of futures without interfering with the process's SPMD flow.

Our current implementation uses Intel Threading Building Blocks' multithreaded execution library [18] for its ease of use and multiple modes of execution. The design of *Global Futures* is not tied to a specific threading package. In fact, we also have a preliminary port of *Global Futures* to Sandia National Laboratories' qthreads multithreaded environment [19], [20].

Global Futures (GF) extends Global Arrays by enabling the execution of user-defined computations (*futures*) on the locations within the distributed memory system that *own* slices of a GA instance. Typically, GA instances are partitioned amongst the processes in a distributed memory system in such a way that each process owns a block or set of blocks of the GA instance. Partitioning of the GA instance can occur in multiple dimensions and can be handled by the GA runtime or specified by the application.

GF is written in C++ and exposes an imperative C-style execution API to the application, with the intent to enable access to the API to Fortran applications as well. GF's functionality is divided into five main categories: initialization, finalization & registration; execution; completion; GA thread safety; and remote data caching. The subsequent sections describe in detail these five categories. Table I summarizes the functionality of the GF API.

1) Initialization, finalization & registration: Global Futures, as any other parallel execution environment for distributed memory systems, must be properly initialized and terminated. User-level functions to be executed as futures must be registered with the system and must follow a particular function prototype. Initialization, finalization and registration are all collective operations requiring agreement by all processes participating in the SPMD execution. All future tasks created for a specific function are called futures of that *type*.

2) Execution: This is GF's principal capability: the ability to remotely execute an user-defined function on the location where a GA instance's array slice resides. GF provides two mechanisms to invoke futures: **active** and **passive**[2]. **Active** futures are intended to execute on the process that owns the data server thread on an SMP node (see Fig. 1) and have direct access to all GA data on that node. **Passive** futures are intended to execute on the process that owns the GA slice according to how the GA instance is distributed. The application will use one mechanism or the other depending on what level of access is needed to the target process's *private* data.

As mentioned previously, we use ARMCI's GPCs to implement the remote invocation of futures. As described, when a GPC is called from a process on one node to a target on another node, the GPC will be executed by the data server thread on the target SMP node. **Active** futures directly utilize this mechanism to create a TBB *enqueued* task upon reception of the GPC call. TBB enqueued tasks are an alternative mechanism to the standard depth-first, Cilk-style TBB task execution mode. Enqueued tasks are actively executed by worker threads upon their appearance in the TBB environment. The GPC execution in the data server thread immediately finishes upon the creation of the TBB enqueued task (it doesn't wait for it to finish). **Active** futures are invoked using the `GFExecute()` API function.

Passive futures require a more complex execution support environment. In this case, when the GPC call is received by the target data server thread, a record is placed onto a Boost library [22] process-based shared memory queue that will be picked up by the target process at a later time. The GPC call then finishes. **Passive** futures are invoked using the `GFEnqueue()` API function.

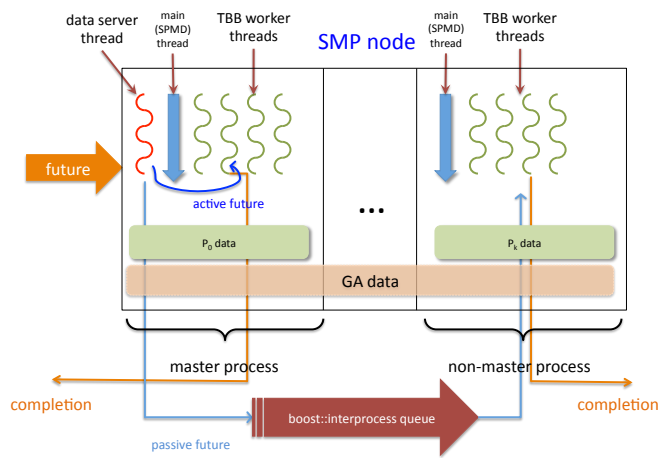

Fig. 2. Execution of futures

In both cases, the application-defined future function can create and invoke more futures. The user-defined future function can also execute arbitrary local and GA operations (via the GF wrapper functions described in Section II-B5). User-defined future functions are not allowed to execute collective operations, including the registration of a new future inside a user-defined future function.

Fig. 2 presents the flow for execution of both **active** and **passive** futures from the perspective of the target SMP node. The future is shown arriving at the node as the orange arrow on the left of the diagram. The data server thread attached to the master process on the node receives the future request as a GPC callback. If it is an **active** future, then it immediately creates a TBB *enqueued* task. This TBB task is then available for execution by the TBB worker threads on the master process. If it is a **passive** future then the data server thread adds an entry to the Boost message queue corresponding to the target process. The passive future will then be picked up for execution by the TBB worker threads on the target process (see Section II-B3 for more details on passive future execution).

When active futures are targeted towards the **same** SMP node as the calling process resides on, then the GPC callback is executed by the calling process **not** by the data server thread. This is correct in the sense that ARMCI's GPC callbacks are intended to execute where the PGAS data is available, thus

[2]This definition differs from the usage in communication runtimes such as MPI [21]

Routine	Collective	Functionality
GFInitialize()	Y	Initializes the *Global Futures* library
GFFinalize()	Y	Finalizes the use of the GF library & releases resources
GFType GFRegister()	Y	Registers a user-defined function for execution as a future
GFHandle GFEnqueue(type, g_a, lo, hi, arg)	N	GF will execute a passive future at the target GA instance location
GFHandle GFEexecute(type, g_a, lo, hi, arg)	N	GF will execute an active future at the target GA instance location
GFWait(hndl)	N	Calling process will block until future is executed
GFQuiesce(type)	N	Calling process will block until all its spawned futures of the type are executed
GFQuiesce()	N	Calling process will block until all its spawned futures are executed
GFAllQuiesce(type)	Y	Calling process will block until all futures of the type spawned by all SPMD processes are executed
GFAllQuiesce()	Y	Calling process will block until all futures spawned by all SPMD processes are executed
GF_Get(), GF_Put(), GF_Acc() and others	N	Wrapper functions for GA thread-safe operations
GF_CachedGet()	N	Blocking get with caching of remote GA data
GF_CachedNbGet()	N	Non-blocking get with caching of remote GA data
GF_CachedNbWait()	N	Finalize non-blocking get operation with caching of remote GA data
GF_CachedAcc()	N	Local element-wise accumulation of data for later accumulation onto GA instance data
GF_CacheReadOnlyEmpty()	N	Flush read-only cached remote data
GF_CacheAccFlush()	N	Element-wise accumulate of locally cached data onto GA instance data

TABLE I
GLOBAL FUTURES API SUMMARY

rendering the calling process on the same SMP node as the data equivalent to the data server thread for this purpose.

3) Completion: GF only provides asynchronous execution semantics for future tasks: that is the execution of a future task happens asynchronously with respect to the invoking process and the process has to explicitly check for completion of that task. GF provides three levels of completion functions for future tasks: one-sided, single future; one-sided, single-source futures; and collective, multi-source futures. In all cases, the GF runtime system signals completion of the future back to the originating process by execution of an ARMCI_Put() on a preallocated PGAS data location.

GFWait(handle) is the API function to detect the completion of a single future. It takes as an argument the future handle returned by a previous call to GFExecute() or GFEnqueue. It will block until the future indicated by the handle has completed.

GFQuiesce() and GFQuiesce(type), respectively, detect the completion of all futures invoked by the calling process or all of futures of certain type invoked by the calling process.

GFAllQuiesce() and GFAllQuiesce(type), respectively, detect the completion of all futures invoked by *all* processes participating in the SPMD execution, or all futures of a certain type invoked by all processes.

GFWait(handle), GFQuiesce() and GFQuiesce(type) are one-sided functions in the sense that only the calling process waits for completion of the respective futures. The target processes where the futures were spawned does not participate, keeping in line with GA's one-sided communication framework.

GFAllQuiesce() and GFAllQuiesce(type) are designed to be called collectively by all processes participating in the SPMD execution. These collective calls have special properties in the context of *Global Futures*: first, **passive** futures that were created for execution on target processes, are actually executed at this time (target processes examine their respective Boost shared memory queues for execution records); second, since all processes are executing the respective AllQuiesce call, no new futures are being created by the main thread of the SPMD processes, which implies that any nested futures being created while inside an AllQuiesce call must be the *residual effect* of previously created futures. All correct *Global Futures* applications should eventually stop creating nested futures and all futures will thus have completed after the AllQuiesce call finalizes.

4) GA remote data caching: One principal advantage of running in a multithreaded execution environment is that the relatively large shared memory within a node can be used in a number of different ways compared to the tighter constraints of a pure process-based SPMD execution environment. GF provides a mechanism for automatically caching GA data read from remote nodes via get() primitives. Remote data read by one thread can then be used by another thread without having to reissue the costly network transfer. We also cache data resulting from Global Arrays' accumulate operations (NGA_Acc(), where a local data buffer is added in an element-wise manner to a size-compatible GA slice.

Cache usage is fully under the control of the application, the programmer must make sure that data is either fully read-only or that there is a single writer. This is similar to the base put/get mechanism in GA's traditional SPMD execution model. GF provides mechanisms to completely flush the read-only caches and to accumulate data to the GA space that has been accumulated locally on the cache.

5) GA thread safety: Global Arrays' current implementation is not thread safe. For this reason, we add a number of wrapper functions with `GF` prefixes (in contrast to GA's `NGA_` and `GA_` prefixes) that utilize a single lock to serialize access under concurrent calls to the GA library. We acknowledge that this solution is not optimal and are actively working to develop thread-safe infrastructure for GA, which can tolerate a greater level of concurrent calls into the library by using finer-grained locking.

III. Self-consistent Field Calculation

We have prototyped the use of *Global Futures* for molecular science applications in the context of the Self-Consistent Field (SCF) calculation [23]. We have selected a standalone SCF application benchmark that is packaged as an example within the Global Arrays software distribution. Within that application, we have focused on the two-electron contribution to the Fock matrix build and have converted it into a *Global Futures*-based computation. This algorithm is the most time-consuming part of this calculation.

The two-electron contribution involves a computationally-sparse n^4 calculation over a n^2 data space. These n^4 tasks need to be enumerated and evaluated. Most of those tasks do not add any significant contribution to the Fock matrix, in fact, only a small percentage of them ($< 1\%$ for larger inputs) do.

The traditional way this calculation is organized in Global Arrays is by having a dynamic counter-based approach in which GA processes contend to obtain the next task identifier from the n^4 pool. The task identifier is used to determine the associated data for the task, which is then inspected to see whether it will add a significant contribution to the Fock matrix. As mentioned before, most tasks do not add a significant contribution to the Fock build. This approach introduces significant additional communication to inspect the insignificant tasks, and does not scale well to large process counts due to the fact that tasks can be assigned to processes in any order without regard for locality. Tasks that do add a contribution then execute a 4-deep loop nest to compute the necessary integrals and contribute to the Fock matrix.

The SCF benchmark uses Global Arrays to store the principal data structures: the Schwarz matrix, the density matrix and the Fock matrix. All of these matrices are distributed equally amongst the running processes using a 2-dimensional distribution, where each process stores a single contiguous block. The SCF application benchmark uses the basis sets for Beryllium atoms.

A. Two-electron Calculation in GF

Each task in the two-electron calculation needs to obtain four equal-sized tiles of data (using `NGA_Get()`): two from the distributed Schwarz matrix and two from the density matrix. The value of the elements of the two tiles from the Schwarz matrix determines whether the task will contribute to the Fock matrix or not. The results are accumulated onto equivalent-sized tiles of the Fock matrix (using `NGA_Acc()`).

With respect to locality, the second tile of the Schwarz matrix and the first tile of the density matrix are co-located and will be owned by the the same process (or set of processes) due to their identical GA distribution. The first tile of the Schwarz matrix and the first tile of the resulting Fock matrix are also co-located.

We describe the different steps to perform the two-electron calculation using *Global Futures*. The overall SCF calculation is an iterative, fixed-point process which continues until certain desired numerical tolerance has been reached. Even though the number of tasks for the two-electron calculation is large (n^4), these tasks do not need to be computed on every iteration of the SCF calculation. It is enough to inspect them and determine which tasks do contribute to the Fock matrix build ("real tasks") and keep track of which tasks from the n^4 space need to be executed by each GA process rank. Thus, the checking and redistribution phases described in Section III-A1 and III-A2 need to be performed only once as part of the first SCF iteration. All the steps in the calculation use **active** futures only.

We use the collective `GF_AllQuiesce()` routine to check for completion of all futures after the checking phase, after the redistribution phase and after the execution phase. We use the collective routine since it also implies a synchronization barrier and guarantees that all GA processes have reached the same step in the computation.

1) Checking phase: In the *Global Futures*-based implementation each process enumerates $\frac{n^4}{p}$ tasks (where p is the number of processes), each process will inspect only tasks for which the first Schwarz tile is local, other tasks are packaged into futures and sent to their home processes for checking. This checking step involves testing the tile to see if *any* of its elements is above the required tolerance level: $\exists\, t \in T, t{\cdot}s \geq \epsilon$, where ϵ is the tolerance level and s is the maximum value of any element of the Schwarz matrix.

This step significantly reduces the amount of communication required since all Global Array accesses become local operations. It also significantly reduces the total number of tasks to execute later (by 94% for a 256 Beryllium atom system).

Tasks that pass this first checking step are then packaged into futures and sent over to the GA process that owns the second Schwarz tile. These tiles are then examined to see if *any* of their elements has an absolute value above the tolerance level: $\exists\ t \in T, \|t\| \geq \epsilon$. The number of tasks sent over for this second round of checking is significantly smaller than the original n^4 set of tasks. Tasks that pass this second check are stored in a local queue on the process that owns the second Schwarz tile.

At the end of this phase, processes will have stored in local queues only tasks that do contribute to the Fock matrix

build. However, the number of tasks per process will be highly unbalanced with respect to other processes due to the inherent sparsity of the task space.

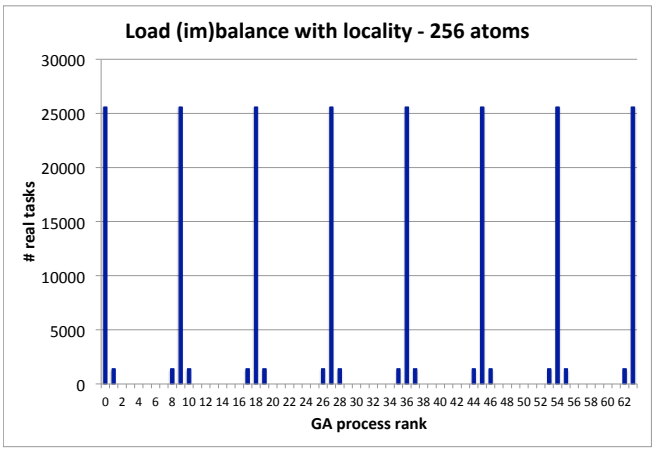

Fig. 3. Load imbalance in the two-electron calculation after task checking

Fig. 3 illustrates the imbalance after the checking phase, by plotting the number of tasks that each GA process rank has in its local queue ("real tasks" that will actually contribute to the Fock matrix build). The plot is for a 256 Beryllium atom system executing on 64 GA processes.

2) Redistribution phase: The tasks stored in the local GA processes' queues are local with respect to the GA operations on the second Schwarz tile (already used in the checking phase) and the first tile of the density matrix. However, as can be seen from Fig. 3, the number of tasks per GA process rank is heavily unbalanced. We rebalance the number of tasks per GA process rank at the cost of reduced locality and increased network communication. Note that all the tasks in the processes' queues are "real" and will contribute to the Fock matrix build. Each task has a similar computational cost (four-deep loop nest), but different communication costs if moved to non-local processes.

We use a `MPI_Allgather()` operation to exchange the number of tasks that each process has and compute a schedule to determine how many tasks need to be sent from processes with a larger number of tasks to those with a lower number of tasks. The task identifiers are packaged into futures and sent to the respective processes for insertion into their local task queues.

3) Execution phase: We now have a fully computationally balanced set of tasks on each GA process rank. Each process uses local futures (essentially local TBB tasks) to compute the contributions of its portion of the task space to the Fock build.

As discussed before, this is the only phase that has to be executed on every SCF iteration. The checking and redistribution phases happen only once.

We use the GF remote data cache to improve performance and reduce communication in the execution phase. We cache the tiles of the Schwarz matrix that each GA process reads on a permanent basis since the matrix is a read-only data structure. We also cache the tiles of the density matrix on a per-iteration basis, that is after every SCF iteration the density matrix cache must be flushed. The density matrix is a read-only data structure in the context of the two-electron routine, however it is updated on every SCF iteration. We also use the accumulate cache for the tiles of the Fock matrix produced by each GA process. The Fock matrix tiles in the cache must be accumulated onto their locations in the Fock GA instance at the end of the two-electron routine.

In our SCF implementation, the available memory per node is sufficient to hold all the data that is being cached. However, for larger problem sizes more sophisticated techniques for reusing space in the remote data cache should be used. Also, our current implementation of the cache is sensitive to the tile sizes being used in the GA data requests, which is appropriate for the two-electron routine in which all requests are of the same size, i.e. array slice specification for `NGA_Get()`s are of the same size. For other algorithms or applications GA data requests might involve array slices of different sizes and thus our current data cache will store duplicate information. A more sophisticated scheme that is not sensitive to tile size needs to be developed.

IV. Experimental Results

We present two sets of experiments: a microbenchmark that indicates the performance potential and quantifies the overhead of *Global Futures*, as well as results on the use of *Global Futures* to implement the two-electron contribution for the SCF application benchmark.

We present results using two platforms: *Trinity*, a 12-node QDR InfiniBand cluster with dual-socket, quad-core Intel Nehalem processors and *Chinook*, a 2,310 node DDR InfiniBand cluster with dual-socket, quad-core AMD Barcelona processors.

A. Microbenchmark

We ran the microbenchmark on *Trinity* to understand the overhead of *Global Futures* and the costs of creating them and checking for completion. The experiment consists of creating 20,000 zero-sized futures on one SMP node and sending them for execution on another node. We use a single process on each SMP node and run the experiment with an increasing number of TBB worked threads on the target node (used to respond to future requests). We reserve one core on each node for the GA data server thread.

Fig. 4 presents the number of futures created per second for both **active** (execute) and **passive** (enqueue) futures. The number of TBB worker threads on the target node increases from one to seven. As can be seen from the graph, **active** futures are significantly faster with a creation time of 8.5 μs per future, while **passive** futures take 86 μs per future. The throughput does not increase significantly as the number of worker threads increases. The lower performance of **passive** futures needs further investigation (and optimization). We believe the current performance is due to inefficiencies in the

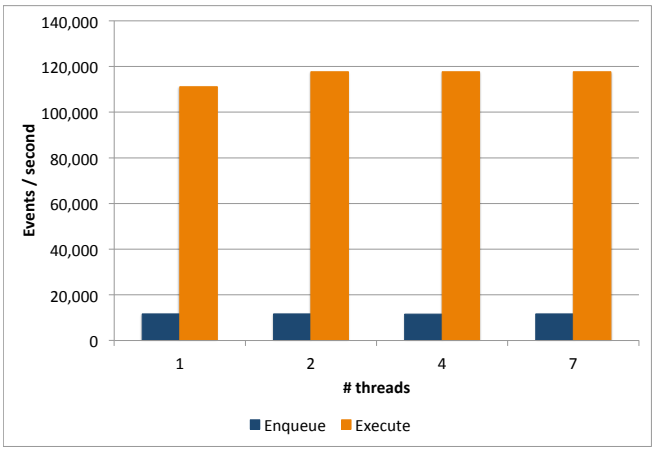

Fig. 4. Futures created per second for both **passive** (enqueue) and **active** (execute) future types

manner that we are using the Boost library shared queue or in its implementation.

B. SCF application benchmark

We executed the full SCF application benchmark on *Chinook* using up to 6,144 cores for two different problem sizes: a 256-atom system and a 352-atom system.

Fig. 5 plots the execution time for the two-electron contribution in SCF on *Chinook* using from 48 cores to 3,072 cores for the 256 Beryllium atom system. We used a single GA process per node (8 nodes to 512 nodes) and 6 TBB worker threads per node. We reserve two cores per node for the GA data server thread and Linux system daemons. We plot the ideal time assuming perfect speedup at 48 cores.

For comparison, we include the performance of a modified version of the dynamic-counter, process-based implementation. We ran this version from 48 to 384 cores, but its lack of scalability prevented us from running larger experiments. We added the use of the remote data cache to this version, as well as a *schedule cache*. During the first iteration of the SCF calculation, each process really goes and grabs a task identifier from the counter until all n^4 tasks are exhausted. Our optimized process-based implementation keeps track of which tasks that the process obtained are "real" (in the sense that they contribute to the Fock matrix build), and reuses this information for the second and subsequent iterations of the SCF calculation. Thus, the cost of dynamically discovering the "real" tasks (using the dynamic counter) is only paid once. In spite of these two optimizations, the scalability is limited due to the very high cost of this dynamic discovery phase. The *Global Futures* implementation avoids this overhead by performing discovery of the "real" tasks almost fully locally (i.e. without GA communication) and establishing the task schedule on each GA process again almost fully locally (futures need to be sent between processes for part of the checking phase and for balancing the load between processes). It is feasible to perform further optimization on the process-based implementation to perform local task discovery and utilize GA or MPI-based communication to exchange tasks for load balance purposes. However *Global Futures* enables a more natural and simpler way of expressing these locality-driven optimizations.

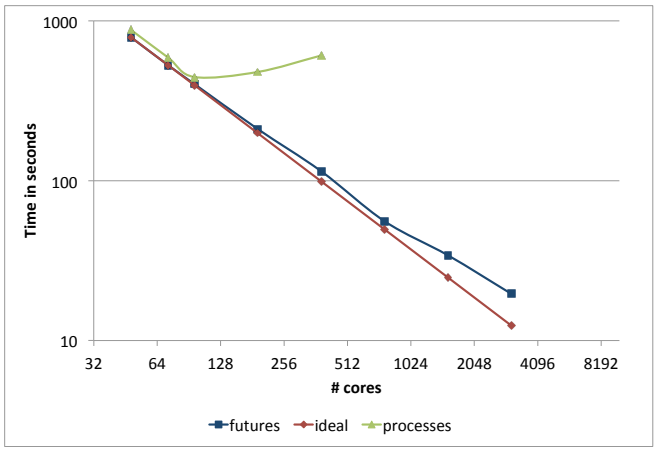

Fig. 5. Execution time on Chinook for the 256-atom system

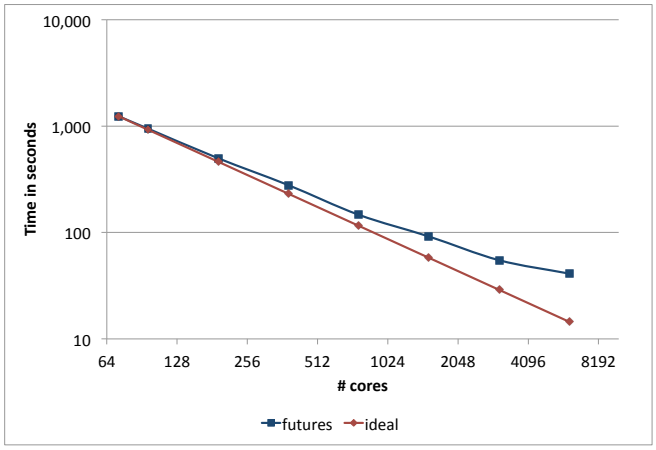

Fig. 6. Execution time on Chinook for the 352-atom system

Fig. 6 plots the execution time for the two-electron contribution to SCF on *Chinook* using from 72 cores to 6,144 cores for the 352 Beryllium atom system. We used a single GA process per node (12 nodes to 1024 nodes) and 6 TBB worker threads per node. We reserve two cores per node for the GA data server thread and Linux system daemons. We plot the ideal time assuming perfect speedup at 72 cores. For this larger input system, we could not run the process-based implementation even at lower core counts.

At larger numbers of cores (3,072 and 6,144) we observe some divergence of the performance compared to the ideal time. We believe this is due to the effect of the single heavy lock protecting concurrent calls into the GA library. At larger core counts, the computation time (number of tasks per process) becomes smaller (19.6 seconds for the 256 atom input, 41 seconds for the 352 atom input) and thus the communication

time becomes a larger fraction of the total time. Since we execute the tasks using local futures to maximize use of the local cores, the GA library needs to be protected. Further work is under way to reduce the cost of thread safety for Global Arrays.

V. Related work

Halstead describes *futures* [24] in the context of the Multilisp programming language. His design focuses on introducing a "future" modifier into the Multilisp type system to indicate that expressions need not be evaluated immediately, thus introducing the possibility of evaluating them concurrently. His design and implementation are tied to the shared-memory semantics of the Multilisp language and are focused on the fine-grained parallelism available in expression evaluation. The *Global Futures* model is designed to work in concert with the Global Arrays PGAS environment on both distributed memory and shared memory systems.

The programming model for the Cray MTA-2 [25] and the Cray XMT [26], [27] provide extensions to the C language to enable the definition and execution of futures in the context of their shared-memory multithreaded execution model. Futures, in this context, are tied to the underlying semantics of the proprietary programming & execution model.

The Charm++ programming model [28] has several features in common with *Global Futures* and Global Arrays including the asynchronous execution of user-defined functions ("chares" in Charm++), detection of quiescence and the use of communication serving threads as part of the runtime system. Charm++ relies on a virtualized execution environment, whereas Global Futures and Global Arrays maintain compatibility with MPI's SPMD execution model.

As mentioned previously, the X10 [8] and Chapel [9] languages implement the APGAS execution model [10] based on the asynchronous execution of activities on nodes in a distributed memory system. Both models require similar capabilities to what *Global Futures* provides in terms of activity creation, management & completion detection.

VI. Conclusions

We have presented the design and implementation of *Global Futures*, an Asynchronous PGAS execution model extension to Global Arrays. We have illustrated its use to enhance the performance and scalability of the two-electron contribution to the Fock matrix in a Self-Consistent Field (SCF) computational chemistry application benchmark.

Global Futures is a library-based model which is fully compatible with Global Arrays and with MPI, as well as fully interoperable with their underlying SPMD execution environment. GF enables the execution of arbitrary user-defined functions on the locations that own array slices of a GA instance. GF provides two modes for execution of futures (**active** and **passive**) as well as multiple mechanisms for detecting completion of those futures.

We have presented results that show how we used GF to increase the scalability of the two-electron contribution to the Fock matrix build in SCF on up to 6,144 cores (1024 dual-socket, quad-core AMD Barcelona nodes) of a DDR InfiniBand cluster. We have also shown that GF's multithreaded execution model has a relatively small overhead compared to the traditional SPMD model when running on the same number of cores for this application, which has a relatively high computation to communication ratio (for real tasks).

Future work includes completing the port of Global Futures to Sandia's `qthreads` library, so that we can compare between different intra-node multithreaded environments. We also plan to experiment with other applications by porting their key kernels to Global Futures, in particular we want to look at more communication-intensive applications in order to optimize the thread-safety mechanisms for Global Arrays. Another interesting aspect would be the comparison of the Global Futures approach to a more traditional hybrid OpenMP/GA multithreaded execution.

References

[1] A. Geist, "Paving the Roadmap to Exascale," *SciDAC Review*, vol. special issue, no. 16, 2010.

[2] D. F. Richards, J. N. Glosli, B. Chan, M. R. Dorr, E. W. Draeger, J.-L. Fattebert, W. D. Krauss, T. Spelce, F. H. Streitz, M. P. Surh, and J. A. Gunnels, "Beyond homogeneous decomposition: scaling long-range forces on massively parallel systems," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. New York, NY, USA: ACM, 2009, pp. 1–12.

[3] 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 *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. New York, NY, USA: ACM, 2009, pp. 1–7.

[4] T. El-Ghazawi and L. Smith, "UPC: unified parallel C," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1188455.1188483

[5] R. W. Numrich and J. Reid, "Co-array Fortran for parallel programming," *SIGPLAN Fortran Forum*, vol. 17, pp. 1–31, August 1998. [Online]. Available: http://doi.acm.org/10.1145/289918.289920

[6] A. Donev, "Rationale for co-arrays in Fortran 2008," *SIGPLAN Fortran Forum*, vol. 26, pp. 9–19, December 2007. [Online]. Available: http://doi.acm.org/10.1145/1330585.1330586

[7] X. Wu and V. Taylor, "Using Processor Partitioning to Evaluate the Performance of MPI, OpenMP and Hybrid Parallel Applications on Dual- and Quad-core Cray XT4 Systems," in *Proceedings of the 2009 Cray Users' Group Meeting*. Atlanta, GA: CUG, May 2009.

[8] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 519–538. [Online]. Available: http://doi.acm.org/10.1145/1094811.1094852

[9] B. Chamberlain, D. Callahan, and H. Zima, "Parallel Programmability and the Chapel Language," *Int. J. High Perform. Comput. Appl.*, vol. 21, pp. 291–312, August 2007. [Online]. Available: http://portal.acm.org/citation.cfm?id=1286120.1286123

[10] G. Bikshandi, J. G. Castanos, S. B. Kodali, V. K. Nandivada, I. Peshansky, V. A. Saraswat, S. Sur, P. Varma, and T. Wen, "Efficient, portable implementation of asynchronous multi-place programs," in *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '09. New York, NY, USA: ACM, 2009, pp. 271–282. [Online]. Available: http://doi.acm.org/10.1145/1504176.1504215

[11] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: a mechanism for integrated communication and computation," in *Proceedings of the 19th annual international symposium on Computer*

architecture, ser. ISCA '92. New York, NY, USA: ACM, 1992, pp. 256–266. [Online]. Available: http://doi.acm.org/10.1145/139669.140382

[12] J. Mellor-Crummey, L. Adhianto, W. N. Scherer, III, and G. Jin, "A new vision for CoArray Fortran," in *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, ser. PGAS '09. New York, NY, USA: ACM, 2009, pp. 5:1–5:9. [Online]. Available: http://doi.acm.org/10.1145/1809961.1809969

[13] J. Nieplocha, B. Palmer, M. Krishnan, and P. Saddayappan, "Overview of the Global Arrays parallel software development toolkit," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1188455.1188690

[14] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà, "Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit," *Int. J. High Perform. Comput. Appl.*, vol. 20, pp. 203–231, May 2006. [Online]. Available: http://portal.acm.org/citation.cfm?id=1125980.1125985

[15] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda, "An evaluation of Global Address Space Languages: Co-Array Fortran and Unified Parallel C," in *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '05. New York, NY, USA: ACM, 2005, pp. 36–47. [Online]. Available: http://doi.acm.org/10.1145/1065944.1065950

[16] J. Nieplocha, V. Tipparaju, M. Krishnan, and D. K. Panda, "High Performance Remote Memory Access Communication: the ARMCI Approach," *Int. J. High Perform. Comput. Appl.*, vol. 20, pp. 233–253, May 2006. [Online]. Available: http://portal.acm.org/citation.cfm?id=1125980.1125986

[17] S. Krishnamoorthy, J. Piernas, V. Tipparaju, J. Nieplocha, and P. Sadayappan, "Non-collective parallel i/o for global address space programming models," in *CLUSTER*, 2007, pp. 41–49.

[18] Intel, "Threading building blocks," http://threadingbuildingblocks.org, Aug. 2011.

[19] K. Wheeler, R. Murphy, and D. Thain, "`qthreads`: An api for programming with millions of lightweight threads," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, april 2008, pp. 1 –8.

[20] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, and J. F. Prins, "Scheduling task parallelism on multi-socket multicore systems," in *Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers*, ser. ROSS '11. New York, NY, USA: ACM, 2011, pp. 49–56. [Online]. Available: http://doi.acm.org/10.1145/1988796.1988804

[21] D. Bonachea and J. Duell, "Problems with using mpi 1.1 and 2.0 as compilation targets for parallel language implementations," *IJHPCN*, vol. 1, no. 1/2/3, pp. 91–99, 2004.

[22] "Boost: C++ libraries," http://www.boost.org, Aug. 2011.

[23] T. P. Hamilton and H. F. Schaefer, "New variations in two-electron integral evaluation in the context of direct SCF procedures," *Chemical Physics*, vol. 150, no. 2, pp. 163 – 171, 1991. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0301010491801263

[24] R. H. Halstead, Jr., "MULTILISP: a language for concurrent symbolic computation," *ACM Trans. Program. Lang. Syst.*, vol. 7, pp. 501–538, October 1985. [Online]. Available: http://doi.acm.org/10.1145/4472.4478

[25] W. Anderson, P. Briggs, C. S. Hellberg, D. W. Hess, A. Khokhlov, M. Lanzagorta, and R. Rosenberg, "Early Experience with Scientific Programs on the Cray MTA-2," in *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2003, p. 46.

[26] J. Feo, D. Harper, S. Kahan, and P. Konecny, "ELDORADO," in *CF '05: Proceedings of the 2nd conference on Computing frontiers*. New York, NY, USA: ACM, 2005, pp. 28–34.

[27] O. Villa, D. Chavarria-Miranda, and K. Maschhoff, "Input-independent, scalable and fast string matching on the Cray XMT," in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, ser. IPDPS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–12. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2009.5161043

[28] L. V. Kale and S. Krishnan, "CHARM++: a portable concurrent object oriented system based on C++," in *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, ser. OOPSLA '93. New York, NY, USA: ACM, 1993, pp. 91–108. [Online]. Available: http://doi.acm.org/10.1145/165854.165874