Scaling Irregular Applications through Data Aggregation and Software Multithreading

Alessandro Morari, Antonino Tumeo
Daniel Chavarría-Miranda
Pacific Northwest National Laboratory
Richland, WA, USA
{*alessandro.morari, antonino.tumeo, daniel.chavarria*}*@pnnl.gov*

Oreste Villa
NVIDIA
Santa Clara, CA, USA
ovilla@nvidia.com

Mateo Valero
Universitat Politecnica de Catalunya
Barcelona Supercomputing Center
Barcelona, Spain
mateo@bsc.es

***Abstract*—Emerging applications in areas such as bioinformatics, data analytics, semantic databases and knowledge discovery employ datasets from tens to hundreds of terabytes. Currently, only distributed memory clusters have enough aggregate space to enable in-memory processing of datasets of this size. However, in addition to large sizes, the data structures used by these new application classes are usually characterized by unpredictable and fine-grained accesses: i.e., they present an irregular behavior. Traditional commodity clusters, instead, exploit cache-based processor and high-bandwidth networks optimized for locality, regular computation and bulk communication. For these reasons, irregular applications are inefficient on these systems, and require custom, hand-coded optimizations to provide scaling in both performance and size. Lightweight software multithreading, which enables tolerating data access latencies by overlapping network communication with computation, and aggregation, which allows reducing overheads and increasing bandwidth utilization by coalescing fine-grained network messages, are key techniques that can speed up the performance of large scale irregular applications on commodity clusters. In this paper we describe GMT (Global Memory and Threading), a runtime system library that couples software multithreading and message aggregation together with a Partitioned Global Address Space (PGAS) data model to enable higher performance and scaling of irregular applications on multi-node systems. We present the architecture of the runtime, explaining how it is designed around these two critical techniques. We show that irregular applications written using our runtime can outperform, even by orders of magnitude, the corresponding applications written using other programming models that do not exploit these techniques.**

I. Introduction

Bioinformatics, complex network analysis, community detection, data analytics, language understanding, pattern recognition, semantic databases and, in general, knowledge discovery are new classes of data-intensive high performance computing applications [10]. These application areas are characterized by dataset sizes already well over the petabyte, which keep growing exponentially.

In addition to being very large, the data employed by these application areas are usually organized in pointer- or linked list-based structures, such as graphs, unbalanced trees or unstructured grids, which exhibit poor spatial and temporal locality, and present fine-grained, unpredictable accesses. This makes these types of codes irregular [29]. The algorithms that process these datasets are also inherently parallel, because they can potentially spawn a concurrent activity for each element (e.g., each vertex or each edge in a graph). Nevertheless, the datasets usually are difficult to partition without generating imbalance among the parallel activities, because their elements are highly interconnected (e.g., power law graphs). Furthermore, they present high synchronization intensity, because the various activities need to access and/or update the same elements.

A key requirement for these applications is that as the dataset size increases, the performance should scale too, maintaining almost constant throughputs. Thus, even if secondary storage technologies, such as Solid State Disks (SSDs), are improving in throughput and access latency, only in-memory processing can provide scalable performance. On the other hand, even if the quantity of memory installable on single node servers and workstations is significantly increasing, only multi-node systems can reach the required memory sizes. However, modern commodity clusters employ processors with advanced cache architectures and rely on data locality, regular computations and bulk communication to achieve high performance. Implementing irregular applications on these machines requires a significant and highly application-specific optimization effort. The distributed memory architecture of these clusters further complicates application development, forcing developers to search for efficient ways to partition datasets and minimize communication overheads.

The Partitioned Global Address Space (PGAS) programming model seems to be a promising solution to develop applications with a shared memory abstraction on distributed memory clusters, without neglecting locality principles. The PGAS data model enables the allocation and access of difficult to partition datasets in the global, aggregate memory of a cluster. However, typically, PGAS remote data access primitives have been designed and optimized for more regular applications and data sets. Multithreading and message aggregation may enable better support for irregular workloads. Multithreading can be effectively used to hide the latencies of memory or network data accesses. For instance, the Cray XMT [13] implements a multi-threaded processor to tolerate memory and network

1530-2075/14 $31.00 © 2014 IEEE
DOI 10.1109/IPDPS.2014.117

access latencies, a scrambled global address space across nodes and full/empty bits associated with each memory word for fine-grain synchronization. However, the reduced market for custom architectures makes them expensive to produce and maintain. Message aggregation (also called message coalescing) allows packing small data requests in a single message, amortizing network overheads and providing higher aggregate network throughput.

In this work, we introduce **GMT** (Global Memory and Threading library), a custom runtime library that enables efficient execution of irregular applications on commodity clusters. To obtain such objective, GMT integrates a PGAS locality-aware global data model together with lightweight software multithreading and message aggregation.

In GMT parallelism is identified through parallel loop constructs, with support for nested parallelism. These constructs enable the expression of the large amount of fine grained parallelism and data accesses typically found in irregular applications. GMT supports millions of lightweight, user-level tasks, mapped on top of a pthreads + MPI software substrate. GMT's runtime implements the key techniques of software multithreading and remote request aggregation.

This work aims at demonstrating that lightweight software multithreading and message aggregation enable GMT to outperform other programming approaches for commodity clusters on large scale irregular applications. We quantify the performance benefits of GMT with respect to MPI with micro-benchmarks performing a large number of fine-grained irregular remote requests. Then, we compare GMT to other PGAS models (UPC on GASNet), to hand-optimized MPI code, and to custom machines (Cray XMT) on a set of typical kernels of large scale irregular applications: Breadth First Search (BFS), Random Graph Walk (RGW) and Concurrent Hash-Map Access (CHMA). Our results demonstrate performance orders of magnitude higher than other solutions for commodity clusters, and competitive performance compared to custom systems. We show that GMT enables high scalability in performance and dataset size, as additional nodes are added to the system.

II. Related Work

Efficiently executing large-scale irregular applications has been a significant research topic for a while.

The Tera MTA and its successors, Cray MTA-2, XMT [13] and Urika are the most relevant examples of custom multi-node supercomputers for irregular applications. They use simple, highly multithreaded (up to 128 hardware threads), cacheless VLIW processors that support a global address space in hardware, interconnected with the high-performance Cray Seastar-2 network. The large number of threads allows tolerating both local and remote memory access latencies. The address space is scrambled across nodes at a fine granularity (64 bytes on the XMT), reducing the hot-spot occurrence and obtaining more uniform data access times. Every memory word is associated with full/empty bits that provide fine grain synchronization. Research has shown that adding remote communication aggregation can provide performance benefits to the architecture with irregular applications [24].

Several types of software approaches have been proposed to enhance the performance of irregular applications on distributed memory machines. Multipol [7] is a library of distributed data structures for irregular problems. The CHAOS/PARTI [12] runtime support library is a set of software primitives that couple partitioners to the application programs, remap data and partition work among processors, and optimize interprocessor communications. These libraries are mainly targeted at optimizing the data partitioning and at reducing communication overheads across processors/nodes through caching mechanisms. They do not implement a global address space, do not exploit multithreading to tolerate data access latencies and do not make use of aggregation to maximize utilization of network bandwidth.

The Partitioned Global Address Space programming model (PGAS) realizes an abstracted shared address space across distributed memory systems, without neglecting data or thread locality. The PGAS programming model is implemented in languages and libraries such as: Unified Parallel C (UPC) [11], [25], [27], Co-Array Fortran (CAF) [16], [23], the Global Arrays (GA) Toolkit [21], X10 [9], Chapel [8] and others. These programming models rely on communication runtime libraries that manage the data exchanges between distributed address spaces, amongst them GASNet [5] and ARMCI [22]. GASNet serves as a communication runtime to several PGAS languages, including Unified Parallel C (UPC) [25], Titanium [28], and Co-Array Fortran [16]. The objective of our work is not propose yet another PGAS programming model or language. GMT is also based around a PGAS concept, but it exposes a very lean Application Programming Interface (API), which provides the basic constructs to access and manage the shared space required by irregular applications. It also supports a very simple form of for loop parallelism. However, differently from other libraries, it couples PGAS with software multithreading and message aggregation, to increase performance and enable multi-node scaling of irregular applications.

Grappa [20] is a runtime which integrates a PGAS programming model and multithreading for latency tolerance, targeted at increasing performance of graph crawling on commodity x86 clusters. Compared to our approach, it employs a substantially different architecture: it is based on GASNet, it has limited support for data aggregation, and it does not exploit thread specialization.

Active Pebbles [26] includes both a programming model for fine-grained data-driven computations and an execution model, which maps the fine-grained expressions to an efficient implementation. Active Pebbles exploits the concept of active messages, and its execution model includes a form of message aggregation. GMT, instead, focuses on providing a simple API targeted to irregular applications, which exploits fine grained loop parallelism and relies on message aggregation and multithreading to increase performance. GMT also exploits thread specialization to realize its functionalities.

Charm++ [17] is a parallel programming system based on a programming model that exploits message-driven objects (*chares*). The runtime can adaptively assign chares to processors during program execution, and supports latency tolerance by switching from blocked processes to others. Again, GMT does not introduce a new programming model. With respect to the Charm++ execution model, our runtime also supports message aggregation and a much finer task granularity for latency tolerance, specifically targeting issues of irregular applications.

High Performance PX (HPX) is a runtime system based on the ParalleX execution model [14]. ParalleX exploits a split-phase multithreaded transaction distributed computing methodology, decoupling computation and communication. ParalleX supports fine-grain multithreading, global address space, overlapping of communication and computation and is able to move work to the data. However, ParalleX is much more complex in scope than GMT, which only addresses issues of irregular applications. Moreover, the current HPX runtime is incomplete, missing data aggregation and developed with an approach to provide features, rather than scalability, first.

There are several libraries for graph processing on distributed systems. Among them, the most widely used are Pregel [19], Giraph [1] and GraphLab [18]. However, they only aim at solving graph traversal and graph-related algorithms, with a specific data structure. GMT, instead, targets a wider class of irregular data structures and algorithms, and aims at providing multi-node scalability through multithreading and data aggregation.

III. Application Programming Interface

Considering the three dimensions of productivity, performance and generality, GMT favors productivity and performance over generality. We employ a programming model that simplifies the development of irregular applications, and rely on the runtime system to enable high performance and scalability for this specific class of applications.

Table I summarizes the primitives currently provided by GMT. In the following sections we describe the characteristics of GMT's API.

A. PGAS data model

The application data is partitioned between global data and local data. The programmer allocates the data structures, mostly arrays, in a virtual global address space, and accesses them through *get* and *put* operations (see Table I). Global data structures are identified by handlers that are passed to the various GMT primitives. Global data is moved into the local space to be manipulated and then written back into the global space. This approach enables the programmer to ignore the actual memory address and cluster node where the data is allocated.

B. Loop parallelism program structure model

In GMT, the programmer expresses parallelism through a parallel loop construct (*gmt_parFor()* in Table I), a parallel control model typical of shared memory paradigms. In contrast with Single Program Multiple Data models, where processes are created at the beginning of a parallel application, this model allows efficiently creating dynamic tasks. In GMT, a *task* is a user-defined function executed for several iterations using the *gmt_parFor()* construct.

The parallel loop construct enables creation of new tasks from iterations of loops over independent individual structure elements (e.g., parallel loops over all vertices or edges of a graph). The application developer can specify how many iterations of the original loop to assign to each task (*chunk_size*), but the runtime is also capable of dynamically detecting if the same processing entity should execute more iterations for load balancing purposes. In the current implementation, the calling task is suspended until all the iterations of the parallel loop are completed. Finally, GMT also supports nested parallel loops, enabling programming patterns such as recursive parallel constructs.

C. Explicit data and code locality management

The GMT allocation primitive allows controlling the data distribution through different strategies. The *GMT_ALLOC_PARTITION* strategy allocates data in a block distributed manner, so that it is uniformly distributed across all the nodes. The *GMT_ALLOC_LOCAL* strategy allocates data only on the memory of the local node. Finally, the *GMT_ALLOC_REMOTE* strategy allocates data on all other nodes except the one that executes the primitive. GMT does not expose the physical location of data to the programmer, to avoid explicit management of data pointers and node ranks. Analogously, GMT task creation policies (*GMT_SPAWN_PARTITION*, *GMT_SPAWN_LOCAL* and *GMT_SPAWN_REMOTE*) control the locality of the tasks created by a parallel loop. The programmer only controls the locality policy, while the runtime takes care of transparently mapping the tasks to the available cluster resources (i.e., processor cores).

D. Blocking and Non-blocking semantics

GMT communication primitives feature both blocking and non-blocking semantics. When using the blocking flavor of the primitives, the task suspends until the operation effectively completes. When the function of a blocking operation returns, the termination of the operation is guaranteed, for both remote and local operations. When using the non-blocking flavor of the primitives, the task continues, and the order of operations is not guaranteed. When the code calls *gmt_waitCommands()*, the task is suspended, until the runtime completes all the pending non-blocking operations. For performance reasons, given the fine-grain nature of communication operations, *gmt_waitCommands()* does not allow waiting for a specific non-blocking operation.

Routine	Functionality
gmt_array gmt_alloc(size, locality)	Allocates a gmt_array with the specified data distribution (partitioned, remote, local)
gmt_free(gmt_array)	Deallocates a previously allocated gmt_array
gmt_putNB(gmt_array, offset, *data, size)	Puts a local buffer into the indicated gmt_array starting at the specified offset (non blocking)
gmt_putValueNB(gmt_array, offset, value, size)	Puts a value into the gmt_array at the specified offset (non blocking)
gmt_getNB(gmt_array, offset, *data, size)	Gets a portion of a gmt_array starting from offset into a local buffer (non blocking)
gmt_waitCommands()	Waits for completion of previously issued non-blocking operations
gmt_put(gmt_array, offset, *data, size)	Blocking put
gmt_putValue(gmt_array, offset, value, size)	Blocking putValue operation
gmt_get(gmt_array, offset, *data, size)	Blocking get
gmt_atomicAdd(gmt_array, offset, value, size)	Atomically adds a value to the value contained in a gmt_array at the specified offset
gmt_atomicCAS(gmt_array, offset, oldValue, newValue, size)	Exchanges a value with the value contained in a gmt_array at the specified offset. Returns the old value
gmt_parFor(tot_iters, chunk_size, *tasks, *args, locality)	Spawn tasks that execute the iterations, up to the total number of iterations, and takes as input the specified argument buffer. Tasks are spawned on all the allocated nodes of the system, locally or remotely, and execute chunk_size iterations per task.

Table I: GMT API summary

E. Explicit synchronization

The programmer explicitly synchronizes access to global data structures. GMT provides atomic operations such as *gmt_atomicCAS()* or *gmt_atomicAdd()* (see Table I), enabling implementation of global synchronization constructs.

IV. Runtime Architecture

We built GMT around three main "pillars": *global address space*, latency tolerance through fine-grained software *multithreading*, and remote data access *aggregation* (also known as coalescing). As previously discussed, global address space support relieves application developers from having to partition data sets as well as having to orchestrate communication. Message aggregation (coalescing) maximizes network bandwidth utilization, despite the small data accesses typical of irregular applications. Fine-grained multithreading enables applications to perform useful work while communication is in progress, hence hiding latencies for remote data transfers as well as the added latency for aggregation.

To explore the design choices of GMT's building blocks, and for the overall experimental evaluation, we employed Pacific Northwest National Laboratory's Olympus supercomputer, listed in the TOP500 [3]. Olympus is a cluster of 604 nodes interconnected through a QDR Infiniband switch with 648 ports (theoretical peak of 4GB/s). Each Olympus' node features two AMD Opteron 6272 processors (codename "Interlagos") at 2.1 Ghz and 64 GB of DDR3 memory clocked at 1600 Mhz. Each socket hosts 8 processor modules (two integer cores, one floating point core per module) on two different dies, for a total of 32 integer cores per node. A module includes a L1 instruction cache of 64 KB, two L1 data caches of 64 KB, and a 2 MB L2 cache. Each 4-module die hosts a shared L3 cache of 8 MB. Dies and processors communicates through HyperTransport.

A. Overview

GMT only targets multi-node distributed memory systems. Indeed, its main features (PGAS, software multithreading and message aggregation) are not needed or have limited utility on a single shared-memory node. GMT targets clusters with multicore x86 processors interconnected with modern, fast networks (e.g., Infiniband, Cray Gemini or Cray Aries). The only software requirements is the support of MPI and Posix threads.

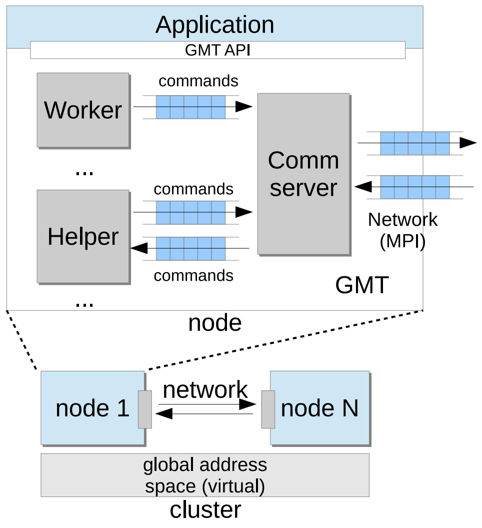

Figure 1: Architecture overview of GMT

Figure 1 illustrates the high level design of GMT. Each node executes an instance of GMT, and the various instances communicate through *commands*. Different types of commands exist for GMT operations, such as global data read/write, synchronization and thread management. Commands may also include data movement (e.g., *gmt_put()* and *gmt_get()*). An instance of GMT, executing in one cluster node, includes three different types of *specialized* threads:

Worker: executes the application code, partitioned in tasks, and generates requests, in form of commands, directed towards both the local node and the remote nodes;

Helper: manages global address space and synchronization, handles incoming requests and generates the related outgoing replies, in form of commands;

Communication Server: communication endpoint on the network, manages incoming and outgoing communication at the node level. Workers and helpers send commands to the communication server, which forwards them to the remote nodes.

A GMT node includes multiple workers and helpers, but only a single communication server. We implemented the

message size	32 proc. no threads	1 proc. 1 thread	1 proc. 2 threads	1 proc. 4 threads
16B	9.63	4.22	2.73	0.77
32B	19.54	9.63	5.70	1.58
64B	39.05	19.54	10.99	3.12
128B	72.26	39.05	19.73	6.22
16KB	2806.94	1924.98	646.52	269.63
32KB	2806.95	2250.15	892.80	469.40
64KB	2815.01	2559.50	794.60	566.09
128KB	2835.98	2709.07	1042.01	564.87

Table II: Transfer rates with in MB/s between two nodes with varying thread and process number.

specialized threads as *POSIX* threads, each pinned to a core.

B. Communication

A fundamental design point for GMT is the choice of the underlying communication library. GMT does not use communication libraries that already provides PGAS primitives, such as GASNet [5], because their implicit communication mechanism does not support message aggregation. Hence, we opted for implementing our own PGAS primitives, designing them with message aggregation from the ground-up. The only requirement is a message passing interface, optimized for high bandwidth. We selected MPI, because it is the de-facto standard for message passing interfaces, and supports the broadest variety of architectures.

We then determined the number of Communication Servers required to maximize node-to-node bandwidth with MPI. We analyzed several combinations of MPI processes and threads per node to determine the highest bandwidth. Table II presents a comparison of the transfer rates between two Olympus' nodes, when transferring a large number of messages from one node to the other and waiting the acknowledgement from the receiver for every 4 messages.

We used a slightly modified version of the OSU Micro-Benchmarks 3.9 [2] (introducing multithreading support) to compare MPI (OpenMPI 1.5.4) with multiple processes (32 on the same node) and MPI (MVAPICH 1.9b) with one, two and four threads. On Olympus, we found multi-threading to have better performance with MVAPICH than with OpenMPI.

MPI with multiple threads per process exhibits low transfer-rates. The best performance is obtained using large messages with 32 processes per node. Nonetheless, we consider this an unfeasible solution for GMT, because of the complexity and the high memory foot-print of managing different address spaces. As shown in Table II, transfer rates are particularly sensitive to the message size. Even if we are showing results for Olympus, we observed similar MPI behavior with other processor architectures and network interconnects. These results drove our decision to design GMT with a single communication server, and to rely on message aggregation to maximize the network bandwidth.

Each worker (or helper) aggregates commands in large messages (buffers), and forwards them to a single communication server that in turn performs the MPI call. The optimal size of the aggregation buffers is a trade off between the bandwidth and the memory foot-print of large buffers. In our experiments with Olympus, we found a buffer size of 64KB

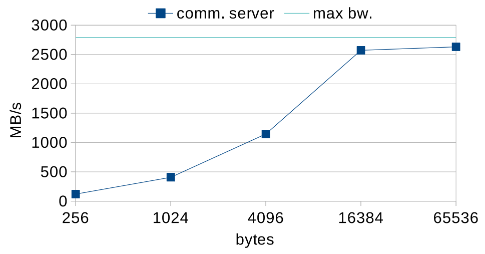

Figure 2: Bandwidth of using a single Communication Server and a single worker with varying message size.

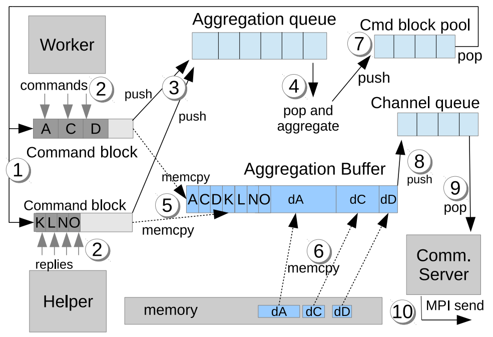

Figure 3: Aggregation mechanism

to be a good compromise. Figure 2 shows the bandwidth reached between two nodes when using one worker and one communication server, while varying the message size. This configuration provides a bandwidth up to 2630 MB/s with 64KB messages, slightly below the measured MPI network bandwidth of 2815.01 MB/s with the same message size.

C. Aggregation

Data aggregation allows efficient exploitation of the available network bandwidth in presence of the fine grained data accesses typical of irregular applications. GMT accumulates commands directed towards the same destination nodes and sends them in bulk. These commands are then unpacked and executed at the destination node.

To increase the opportunity of aggregating network transfers, GMT uses *aggregation queues* to collect request or reply commands with the same destination from all the workers and helpers of a node. To this aim, GMT employs high-throughput, non-blocking concurrent aggregation queues. These aggregation queues support concurrent access from multiple workers and helpers, but the cost of concurrent accesses to the queues is too high (high access frequency) if performed for every generated command. For this reason, GMT uses a pre-aggregation phase: each worker (or helper) initially collects commands in local *command blocks* and then inserts them into the aggregation queue.

Figure 3 describes the aggregation mechanism in GMT. When a worker or a helper thread starts generating commands, it requests one of the pre-allocated command blocks from the command block pool (1). Command blocks are reusable arrays containing several commands. They are pre-allocated and recycled for performance reasons. While a worker executes the application code, it generates commands of various types that are collected into the local command block (2). Helpers also generate commands when creating replies for incoming operations. In the example, a worker generates commands A,C,D and a helper generates commands K,L,N and O. Waiting until the command block is full may increase too much the latency. For this reason, workers or helpers push command blocks into the an aggregation queue (3) when one of the following conditions is verified: i) the command block is full; ii) the command block has been waiting longer than a predetermined time interval (clock cycles). A command block is considered full when all the available entries are occupied with commands, or when the size in bytes of the commands, with the attached data, reaches the maximum size of the aggregation buffer.

Aggregation queues are shared among all the workers and helpers in the node, and there is one of them for each destination node. The aggregation consists in copying commands into an *aggregation buffer* that is sent over the network to the destination node. When a worker (or a helper) finds that the aggregation queue for a destination node has enough command blocks to fill an aggregation buffer (in terms of number of commands or in equivalent byte size), the actual aggregation starts. Aggregation can also start because the aggregation queue has been waiting longer than a predetermined time interval. When aggregation starts, the worker (or helper) pops command blocks from the aggregation queue to fill the aggregation buffer (4). GMT uses a fixed pool of aggregation buffers that are recycled to save memory space and eliminate allocation overhead. Multiple commands are copied from their command block into the aggregation buffer at once (5). For commands that require data movement (such as *gmt_put()* or the reply to a *gmt_get*), the data is also copied from the memory into the aggregation buffer (6). In the example in Figure 3, the data for the commands A, C, D is represented as dA, dC and dD respectively. After the copy, commands blocks are returned to the command block pool (7). The aggregation algorithm continues to push command blocks until an aggregation buffer is full. When this happens, the worker (or helper) pushes the aggregation buffer into a channel queue (8). Channel queues are high-throughput single-producer single-consumer queues that enable the communication between a worker (or helper) and the communication server.

The communication server continuously polls the channel queues, checking if new filled aggregation buffers are available. If so, the communication server pops a filled aggregation buffer and performs a non-blocking MPI send. It then returns the aggregation buffer into the pool of available aggregation buffers (not represented in the figure).

D. Multithreading

Concurrency, through fine-grained software multithreading, allows tolerating the added latency for aggregating communication operations. We use the term *task* to identify a function pointer and an execution context inside GMT, while we use the term specialized thread (or, simply, thread) to identify either a worker, a helper or the communication server. Each worker executes a set of GMT tasks. The worker switches among tasks' contexts every time it generates a blocking command that requires a remote memory operation. The task that generated the command executes again only when the command itself completes (i.e., it gets a reply back from the remote node). In case of non-blocking commands, the task continues executing until it encounters a *gmt_wait_commands()* primitive.

ctxt switches	**1 task**	**8 tasks**	**64 tasks**	**1024 tasks**
1	1816.00	1500.25	1536.81	1799.10
100	497.31	496.71	554.25	590.91
1000	517.14	494.56	545.00	579.13

Table III: Latency (clock cycles) of a context switch when increasing the number of tasks and the number of context switches per task.

GMT implements custom context switching primitives that avoids some of the lengthy operations (e.g., saving and restoring signal mask) performed by the standard *libc* context switching routines. To evaluate the maximum network latency that is potentially tolerable, we measured the cost of context switching among two or more tasks. We performed an experiment that executes an increasing number of context switches among an increasing number of tasks.

Table III shows the latency, in clock cycles, to execute a context switch with this experiment. When increasing the number of context switches from 1 to 1000, we observe the effect of the caches that avoids retrieving the task context from memory. We also observe that the latency only slightly increases when increasing the number of tasks.

Given the typical network latency being in the order of 10^6 cycles and a single context switch being less than 10^3 cycles, GMT can perform more than 10^3 context switches during the time a task is waiting for a remote reply. The optimal number of concurrent tasks per worker actually depends on the architecture (cache size) and on the workload (amount of work per task).

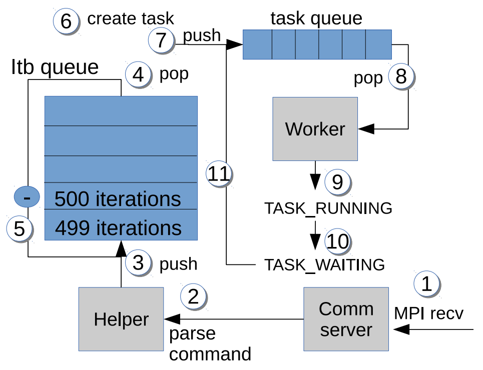

Figure 4: Fine grain multithreading in GMT.

In GMT, the programmer typically generates tasks (except the task zero) by calling the *gmt_parFor()* construct. Figure 4 schematically shows how GMT executes a task. A node

Parameter	Configuration
NUM_WORKERS	15
NUM_HELPERS	15
NUM_BUF_PER_CHANNEL	4
MAX_NUM_TASKS_PER_WORKER	1024
SIZE_BUFFERS	65536

Table IV: GMT configuration parameters for Olympus

receives a message containing a *spawn* command (1) that a worker in a remote node generated when encountering a *gmt_parFor()* construct. The communication server passes the buffer containing the command to an helper, which parses it and executes the command (2). The helper then creates an *iteration block* (itb). The itb is a data structure that contains the function to execute, the arguments of the function itself, and the number of tasks that execute the same function. Each task represents a single iteration of the original *parFor*. This way of representing a set of tasks avoids the cost of creating a large number of function arguments and sending them over the network. In the following step, the helper pushes the iteration block into the itb queue (3). Then, an idle worker pops the iteration block from the itb queue (5), decreases the counter of the iterations of t and pushes it back into the itb queue (6). The worker creates t tasks (6) and pushes them into its private task queue (7). An idle worker pops a task from its task queue (8). If the worker can execute the task (i.e., all remote requests are completed), it restores the task's context and executes it (9). Otherwise, it pushes the task back into the task queue. The task contains user-level application code, which eventually calls one of the GMT primitives. In case the GMT primitive is a blocking remote request (e.g., *gmt_get()*), or an explicit wait (*gmt_waitCommands()*), and they are not completed, the task enters into a waiting state (10) and is reinserted into the task queue for future execution (11).

V. Experimental Evaluation

As introduced in section IV, we evaluated GMT on PNNL's Olympus supercomputer. GMT can adapt to other systems by tuning configuration parameters defined at installation time. For this work, we empirically optimized the parameters for the Olympus system. Table IV presents the configuration used in our benchmarks.

Parameters *NUM_WORKERS* and *NUM_HELPERS* define the number of worker and helper threads per node. Parameters *NUM_BUF_PER_CHANNEL* and *SIZE_BUFFERS* determine how many buffers are allocated to each communication channel and their size in bytes. Finally, *MAX_NUM_TASKS_PER_WORKER* defines the maximum number of tasks concurrently executed by each worker. A detailed explanation of the effects and correlations of each parameter is outside the scope of this work.

A. Micro-benchmarks

In this section we characterize the peak communication performance of GMT. The aim of this characterization is to quantify the effects of the aggregation when performing a large number of basic GMT remote operations. When GMT executes a series of fine-grain put operations, we expect to observe a considerable performance improvement in bandwidth utilization with respect to sending MPI messages of the same size, because of aggregation. Furthermore, increasing the number of concurrent tasks increases the likelihood of generating communication operations. Thus, we expect that aggregate bandwidth increases with the number of concurrent tasks in the node.

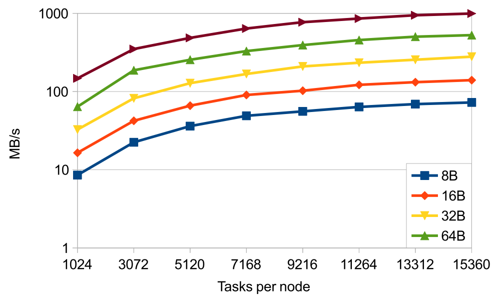

Figure 5: Transfer rates of put operations between 2 nodes while increasing concurrency

Figure 5 shows how transfer rates between two nodes behave when increasing the number of tasks per node in GMT. Every task executes 4096 blocking put operations. All the experiments use 15 workers, but we increase the number of tasks for the node. The graph plots message sizes from 8B to 128 bytes. We verify that increasing the concurrency in the node increases the transfer rates, because there is a higher number of messages that GMT can aggregate. With 1024 tasks, puts of 8 bytes reach a bandwidth of 8.55 MB/s. With 15360 tasks, the bandwidth increases to 72.48 MB/s, a factor of 8.4. Larger messages provide higher bandwidth, because they reduce network overhead. With messages of 128 bytes and 15360 tasks, GMT reaches almost 1 GB/s, while the best MPI implementation reaches 72.26 MB/s (using 32 processes). At these message sizes, with blocking operations, the task switching time also becomes a factor. In fact, a node should be able to generate as many network references as possible to saturate the effective network bandwidth for small messages. When concurrent tasks emit communication operations in parallel, they increase the injection rate. However, if the task switching time is too high, there is an added latency between an injected network operation and another, which may not allow maximizing network utilization, even considering the overheads for packet headers.

With more destination nodes, the probability of aggregating enough data to fill a buffer for a specific remote node decreases. To verify the behavior of aggregation with higher numbers of nodes, we executed the same experiment on 128 nodes. Figure 6 shows the results. If we compare it to the previous figure, we observe a slight degradation in performance. However, aggregation is still very effective with respect to MPI send operations of the same size. For instance, GMT with messages of 16 bytes over 128 nodes reaches a bandwidth of 139.78 MB/s, versus the 9.63 MB/s of the MPI send operation (using 32 processes).

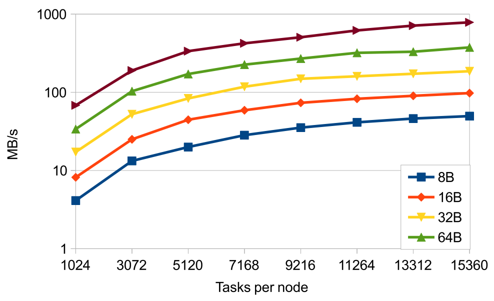

Figure 6: Transfer rates of put operations among 128 nodes (one to all) while increasing concurrency

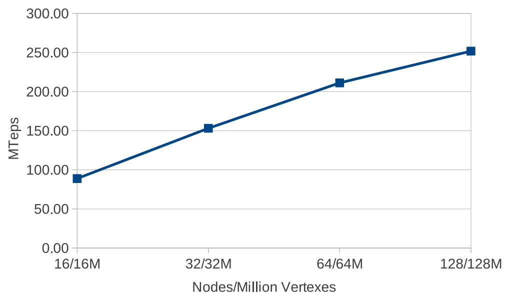

Figure 7: Million traversed edges per second for the GMT implementation of the BFS (weak scaling)

B. BFS

BFS is one of the most common graph search kernels, and a building block for many graph analysis applications. As a matter of fact, the BFS is part of the Graph500 [15] benchmark suite and a de-facto benchmark for irregular applications. All the implementations exploit parallelism on the vertex queue while progressing through the various levels of exploration. The codes for GMT and Cray XMT [4] are essentially identical, except that the proprietary Cray XMT primitives are substituted with GMT primitives. The code for UPC, instead, uses several optimizations, such as caching the exploration map, aggregating communication at the application code level and using asynchronous gets for the aggregated transfers. The complexity of the UPC version accounts for ≈ 700 lines of code, compared to ≈ 80 of the other implementations.

Figure 7 shows the weak scaling of the GMT implementation, measured in million of traversed edges per second. The implementation performs single-word memory accesses on the global graph structure. For this experiment we used randomly generated graphs, increasing the size of the graph of 1 million vertices for each node added. Each vertex has at most 4000 edges connecting to random vertices in the graph. Therefore, the largest graph on the 128 nodes configuration has 128 million vertices and 258 billion edges, for a total memory footprint of ≈ 2 TB.

Figure 8 shows the strong scaling of the GMT implementation, comparing it to the equivalent queue-based implementations for UPC and the Cray XMT. For these experiments, we used a random graph of 10 million vertices and 2.5 billion edges, due to the maximum memory capacities of the platforms (1 TB on the Cray XMT). The UPC implementation does not scale in performance, and experiments with more than 16 nodes did not complete in reasonable times. An interesting consideration is the amount of parallelism necessary to fully utilize the various platforms. While the Cray XMT needs 128 threads per processor, GMT requires 1024 user tasks per worker. With 128 nodes and 15 workers per node, GMT needs 2 million tasks to fully utilize the system. Indeed, GMT's performance starts to decrease above 64 nodes, because the available parallelism in the application is not enough.As the graph in Figure 8 shows, the BFS implementation for GMT outperforms the other implementations. The programming effort for GMT is very similar to the one for the Cray XMT, while the UPC version was significantly more challenging to implement and optimize.

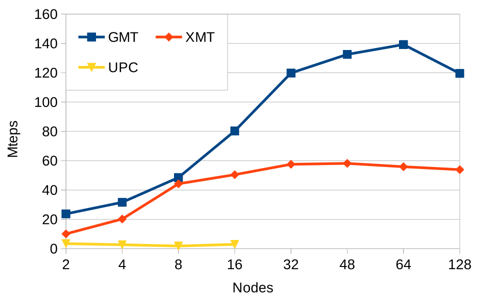

Figure 8: Million traversed edges per second for the BFS implementation on GMT, UPC and Cray XMT (strong scaling)

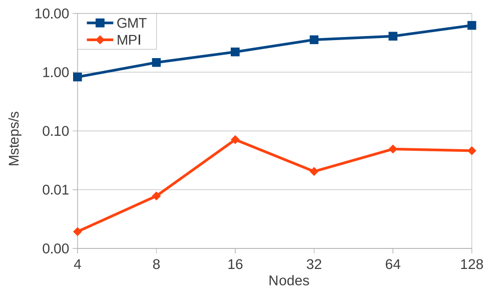

Figure 9: Millions of steps per second for the random walk implementation on GMT and MPI (weak scaling)

C. Graph Random Walk

Graph Random Walk (GRW) randomly traverses a graph with the purpose of collecting vertex/edge information or of understanding graph properties. Many application areas, such as artificial intelligence, brain research and game theory, exploit the GRW kernel in their algorithms. In a GRW, each task starts from a *source* node, chooses randomly a neighbor to visit, and continues the walk until it has visited L connected nodes. Our code, given a connected graph of V vertices and E edges, assigns $V/2$ vertices as *source* nodes to $V/2$ parallel tasks. Each task performs a walk of length L. Implementing the GRW in GMT is fairly simple: (i)

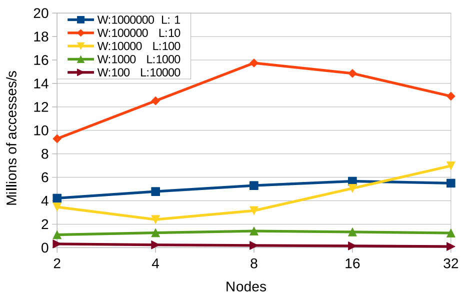

Figure 10: Number of strings hashed and inserted per second (Millions of accesses/s) for the GMT implementation of the Concurrent Hash Map Access benchmark. In the legend, W refers to the number of tasks and L to the number of accesses performed by each task.

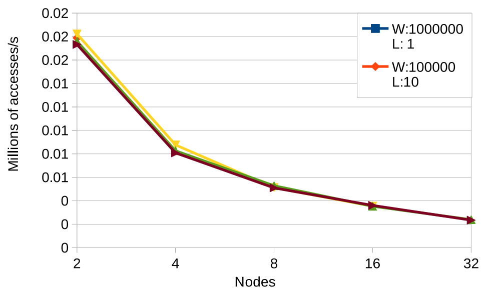

Figure 11: Number of strings hashed and inserted per second (Millions of accesses/s) for the MPI implementation of the Concurrent Hash Map Access benchmark. In the legend, W refers to the number of processes and L to the number of accesses performed by each process.

gmt_parfor() spawns $V/2$ tasks; (ii) each task performs a random walk of length L, accessing the graph with GMT primitives. We compare the GMT implementation to a state-of-the-art MPI implementation employed in fast matching algorithms [6]. This approach, rather than making a process retrieve non-local data, delegates the completion of a walk to the process that locally owns the data. The algorithm employs P processes, divided as one master process and $P-1$ slave processes. Given a graph with V vertices and E edges, the algorithm performs the following steps: (1) the master process initializes and distributes V/P vertices to all P processes (including itself); (2) each process starts $V/(P*2)$ walks ($V/2$ walks in total for the whole system) of length L from each one of its assigned vertices; (3) if a vertex v is not owned by the current process, it delegates the process owning v to continue the walk; (4) when a process completes all its local walks, it communicates to the master the number of completed walks (i.e., walks that traversed L nodes) and waits for walks to continue from other processes (in case there are new walks, it restarts from step 3); (5) when all the $V/2$ walks are completed, the master sends the quit command to the slave processes. The MPI algorithm exploits message aggregation to reduce fine-grain communication. Whenever a process requires the delegation of walks to other processes, it buffers all the requests for each process and sends them out at once only after completing the local walks (i.e., end of step 4). To empirically quantify the complexity of the two implementations, we measured the source lines of code for the GMT and MPI implementations. We found that the source code of the MPI version is a factor of 15 longer than the GMT version. Figure 9 shows (in logarithmic scale) the performance of the GRW, measured in million of traversed edges per second (MTEPS). The experiments use a randomly generated graph of one million vertices per-node (weak scaling) with an average of 4000 edges per vertex. The figure shows that GMT is one or more orders of magnitude faster than the MPI implementation.

D. *Concurrent Hash Map Access*

Concurrent Hash Map Access (CHMA) is a synthetic benchmark where multiple concurrent activities access a hash map to check the presence of a hashed element (e.g., a string or a signature). If the element is found, it is modified according to a predetermined rule and stored back into the hash map. The behavior of this kernel is typical of streaming applications such as virus scanning, spam filters, natural language processing, and of information retrieval applications that need to store, filter and manipulate large amounts of streaming data. In our experiments, we used a pool of 100 million strings with at most 20 characters each to populate a hash map of 10 million entries. After the initialization, W concurrent tasks perform the following operations for 'L' steps: (1) start from a given input string; (2) find if it is present in the hash map; (3) if it is present, perform a string reverse operation; (4) hash the new string and store it back in the hash map; (3) if it is not present, get a new input string. We compare both an MPI and a GMT implementation. In the MPI implementation, each MPI rank is responsible for a portion of the hash map. Only the process that owns the related portion of the hash map checks and inserts the strings. However, if the current process does not own the hashed string, it sends the string to its owner. Small MPI messages are very frequent, because a process cannot proceed with a new string until it has finished manipulating the previous one. It is possible to implement partial caching with remote bulk updates, but it requires employing expensive checks and invalidation mechanisms. On the other hand, the GMT implementation is straight forward: the *gmt_parfor()* construct spawns W tasks, each task independently performs get/put and atomic compare and swap operations on the hash map for L steps. As for the other two kernels, the MPI solution for CHMA was significantly more complex and difficult to implement than the GMT code.

Figures 10 and 11 respectively show the throughput, in million of strings hashed and inserted per second (Millions of accesses/s) of the GMT and the MPI implementations, while increasing the number of cluster nodes, varying the number of tasks (or processes) that concurrently access the hash map (W) and the number of steps (L) performed by each tasks (or process). The performance between the GMT and the MPI implementations differs by two or more orders of magnitude, because of the fine grained communication involved in the kernel.

VI. Conclusions

We presented **GMT**, a Global Memory and Threading library that enables efficient execution of irregular applications on commodity clusters. GMT integrates a PGAS data substrate with simple loop parallelism. It provides a simple interface for designing applications with large, irregular data structures, without requiring data partitioning. GMT is built around the concepts of lightweight user level multithreading and data aggregation to reduce the impact of fine grained, unpredictable data accesses typical of irregular applications. GMT tolerates network communication latencies by switching thousands of tasks on each available worker thread. GMT implements multi-level aggregation to maximize network bandwidth utilization with small messages. GMT aims at providing a solution to scale irregular applications in performance and size by adding more nodes to a cluster. We characterized the communication performance of GMT, and compared it to hand-optimized UPC and MPI code, as well as to custom machines designed for irregular applications, on a set of typical large-scale, irregular application kernels. We demonstrated speed ups of orders of magnitude compared to other programming approaches for commodity clusters, and performance comparable to custom machines.

References

[1] Apache Giraph. http://incubator.apache.org/giraph/.

[2] OSU Micro-Benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/.

[3] TOP500 - PNNL's Olympus entry. http://www.top500.org/system/177790.

[4] D. A. Bader and K. Madduri. Designing Multithreaded Algorithms for Breadth-First Search and st-connectivity on the Cray MTA-2. In *ICPP '06: the 2006 International Conference on Parallel Processing*, pages 523–530, 2006.

[5] D. Bonachea. Gasnet specification, v1.1 - t.r. csd-02-1207. Technical report, UC Berkeley, October 2002.

[6] U. V. Catalyurek, F. Dobrian, A. Gebremedhin, M. Halappanavar, and A. Pothen. Distributed-memory parallel algorithms for matching and coloring. In *IPDPSW '11: the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, pages 1971 – 1980, 2011.

[7] S. Chakrabarti and K. Yelick. Implementing an irregular application on a distributed memory multiprocessor. In *PPOPP '93: the 4th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 169–178, 1993.

[8] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, 2007.

[9] 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 *OOPSLA '05: the 20th annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538, 2005.

[10] J. Chen, A. Choudhary, S. Feldman, B. Hendrickson, C. Johnson, R. Mount, V. Sarkar, V. White, and D. Williams. Synergistic challenges in data-intensive science and exascale computing. Doe ascac data subcommittee report, March 2013.

[11] G. Cong, G. Almasi, and V. Saraswat. Fast PGAS connected components algorithms. In *PGAS '09: the 3rd Conference on Partitioned Global Address Space Programing Models*, pages 13:1–13:6, 2009.

[12] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *J. Parallel Distrib. Comput.*, 22(3):462–478, Sept. 1994.

[13] J. Feo, D. Harper, S. Kahan, and P. Konecny. ELDORADO. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pages 28–34, 2005.

[14] G. Gao, T. Sterling, R. Stevens, M. Hereld, and W. Zhu. Parallex: A study of a new parallel computation model. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–6, 2007.

[15] The graph 500 list. http://www.graph500.org, April 2013.

[16] G. Jin, J. Mellor-Crummey, L. Adhianto, W. Scherer, and C. Yang. Implementation and Performance Evaluation of the HPC Challenge Benchmarks in Coarray Fortran 2.0. In *IPDPS '11: IEEE International Parallel and Distributed Processing Symposium*, pages 1089 –1100, 2011.

[17] L. V. Kale and S. Krishnan. *CHARM++: a portable concurrent object oriented system based on C++*, volume 28. ACM, 1993.

[18] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012.

[19] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD '10: ACM International Conference on Management of data*, pages 135–146, 2010.

[20] J. Nelson, B. Myers, A. H. Hunter, P. Briggs, L. Ceze, C. Ebeling, D. Grossman, S. Kahan, and M. Oskin. Crunching large graphs with commodity processors. In *HotPar '11: the 3rd USENIX conference on Hot topic in parallelism*, HotPar'11, pages 10–10, 2011.

[21] 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.*, 20(2):203–231, 2006.

[22] 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.*, 20(2):233–253, 2006.

[23] R. W. Numrich and J. Reid. Co-arrays in the next Fortran Standard. *SIGPLAN Fortran Forum*, 24(2):4–17, Aug. 2005.

[24] A. Tumeo, S. Secchi, and O. Villa. Designing next-generation massively multithreaded architectures for irregular applications. *Computer*, 45(8):53–61, 2012.

[25] UPC Consortium. UPC Language Specifications v. 1.2. www.gwu.edu/ upc/docs/upc_specs_1.2.pdf, May 2005.

[26] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine. Active pebbles: parallel programming for data-driven applications. In *ICS '11: the International Conference on Supercomputing*, pages 235–244, 2011.

[27] S. Xu and L. Chen. Shared work list: hacking amorphous data parallelism in UPC. In *PMAM '12: the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 124–133, 2012.

[28] K. Yelick, P. Hilfinger, S. Graham, D. Bonachea, J. Su, A. Kamil, K. Datta, P. Colella, and T. Wen. Parallel Languages and Compilers: Perspective from the Titanium Experience. *Int. J. High Perform. Comput. Appl.*, 21(3):266–290, Aug. 2007.

[29] K. A. Yelick. Programming models for irregular applications. *SIGPLAN Not.*, 28:28–31, January 1993.