GMT: Enabling Easy Development and Efficient Execution of Irregular Applications on Commodity Clusters

Alessandro Morari*†, Oreste Villa‡, Antonino Tumeo*,
Daniel Chavarría-Miranda*, Mateo Valero†
*Pacific Northwest National Laboratory - Richland, WA, USA
†Universitat Politecnica de Catalunya - Barcelona, Spain
‡NVIDIA - Santa Clara, USA

1. INTRODUCTION

Emerging high performance applications for bioinformatics, big science, complex network analysis, community detection, data analytics, language understanding, pattern recognition, semantic databases and, in general, knowledge discovery present dataset sizes well over the petabyte scale and exponentially growing. Only multi-node systems may allow in-memory processing of their datasets. However, these applications exploit pointer- or linked list-based data structures such as graphs, unbalanced trees or unstructured grids which exhibit poor spatial and temporal locality and are difficult to partition on distributed memory systems [2]. They also have fine-grained data accesses that, paired with the unpredictability, lead to low utilization of the available memory and network bandwidth of modern clusters. These applications are inherently parallel, because they can spawn concurrent activities for each element of their datasets, but they also present high synchronization intensity.

In this poster we introduce **GMT** (Global Memory and Threading library), a custom runtime library that enables efficient execution of irregular applications on commodity clusters. GMT only requires a cluster with x86 nodes supporting MPI. GMT integrates the Partititioned Global Address Space (PGAS) locality-aware global data model with a fork/join control model common in single node multithreaded environments. GMT supports lightweight software multithreading to tolerate latencies for accessing data on remote nodes, and is built around data aggregation to maximize network bandwidth utilization.

2. PROGRAMMING MODEL AND API

Figure 1 summarizes GMT's Application Programming Interface (API). GMT provides a simple way to program large-scale irregular applications on commodity clusters, without neglecting performance. To reach this objective, the runtime implements a PGAS data model across the cluster. This relieves application developers of the partitioning of the data structures across the nodes of the cluster. The programmer allocates the data structures, mostly arrays, in a virtual global address space, and accesses them through *get* and *put* operations. As in other PGAS approaches, GMT also offer the opportunity to control the allocation policy, maximizing locality exploitation. The programmer can allocate data in a partitioned way, uniformly distributed across all the nodes of the system, or locally, on the memory of the node where the code is executing. The programmer can also allocate data on all the nodes except the one where the code is executing. In irregular applications, parallelism mostly resides in loops (e.g., loops on all vertices or edges of a graph). GMT provides a parallel fork construct, which allows spawning different iterations of the loops as independent tasks. GMT also supports nested parallelism (any task can spawn other tasks). Developers also have direct access to constructs for atomic compare and swap and atomic addition on individual elements of the global memory address space. Load balancing is completely transparent. The runtime takes care of distributing tasks across nodes, and on the cores available on each node. GMT also supports task locality: the programmer can spawn tasks that can execute on any node of the system, only on the local node or only on remote nodes.

3. GMT ARCHITECTURE

We built GMT around three main "pillars": *global address space*, latency tolerance through fine-grained software *multithreading*, and *aggregation*. Figure 1 illustrates the (high level) architecture of GMT. GMT realizes a virtual global address space across the nodes of the cluster. Each node executes an instance of GMT. An instance of GMT includes three different types of *specialized* threads, as follows: *i. Workers*: execute the application code, partitioned in tasks; *ii. Helpers*: manage global address space and incoming communication; *iii. Communication server*: entry point to the network, manages incoming and outgoing communication at the node level with MPI. There are multiple workers and helpers, but only one communication server per node. GMT instances communicate through *commands*. The commands include data accesses, synchronization and thread management operations. They may also have data attached. Workers only generate commands, while helpers receive and generate commands.The communication server transfers commands from one node to the other through MPI. When a task executes a data access or a synchronization operation on a remote memory location, the worker generates the corresponding command and switches its execution context to another task in its own private queue. This allows tolerating network latency. To reduce overheads for fine-grained network transactions, GMT aggregates the commands directed towards a node before sending them out. GMT supports two level of aggregation. The first one happens at the level of the specialized threads. Before sending out commands, workers and helpers accumulate them in different arrays depending on their destination node. When an array is full in terms of number of commands or in byte equivalent size

Routine	Functionality
gmt_array gmt_alloc(size, locality) gmt_free(gmt_array)	Allocates a gmt_array with the specified locality (partitioned, remote, local) Memory freeing
gmt_putNB(gmt_array, offset, *data, size) gmt_putValueNB(gmt_array, offset, value, size) gmt_getNB(gmt_array, offset, *data, size) gmt_waitCommands()	Puts a local array into the gmt_array starting at the specified offset (non blocking) Puts a value into the gmt_array at the specified offset (non blocking) Gets a portion of a gmt_array starting from offset into a local array (non blocking) Waits completion of previous non-blocking operations
gmt_put(gmt_array, offset, *data, size) gmt_putValue(gmt_array, offset, value, size) gmt_get(gmt_array, offset, *data, size)	Blocking version of the put Blocking version of the put of a value Blocking version of the get
gmt_atomicAdd(gmt_array, offset, value, size) gmt_atomicCAS(gmt_array, offset, oldValue, newValue, size)	Atomically adds a value to the value contained in a gmt_array at the specified offset 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 executes the number of loop iterations, up to the total number of iterations, and takes in input the specified arguments. Tasks are spawned on all the nodes of the system, locally or remotely, and execute chunk_size iterations each.

Table 1: GMT API summary

(commands may have data attached), or after a predefined time interval (to guarantee a higher bound for the maximum latency of a memory operation), the thread moves the data in the shared aggregation queues for the whole node. There are shared aggregation queues for each one of the possible destination nodes. When one of the specialized threads finds that the aggregation queue it is writing to is full (because of the number of commands, or the equivalent byte size), or the predefined time interval for aggregation has passed, it prepares the data and copies them in one of the buffers that the communication server uses to send out data.

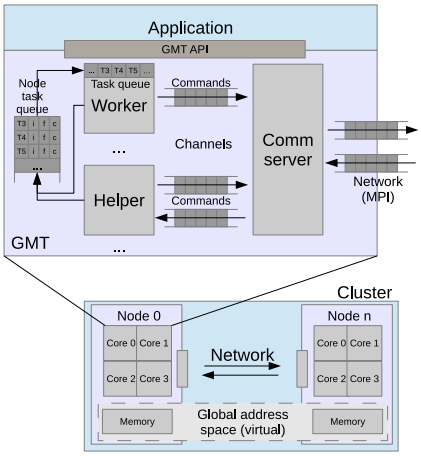
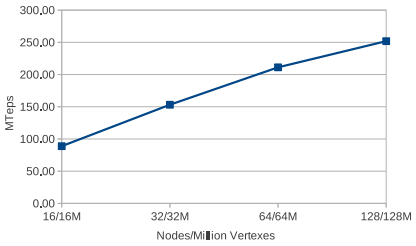


Figure 1: GMT architecture

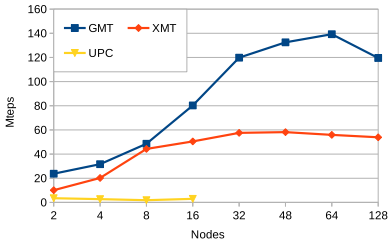
4. EVALUATION

We evaluated GMT on the Pacific Northwest National Laboratory's Olympus supercomputer [1]. Olympus is a cluster of 604 nodes interconnected through QDR Infiniband. Each Olympus' node features two AMD Opteron 6272 ("Interlagos" at 2.1 GHz for a total of 32 integer cores) and 64 GB of DDR3 memory. We run GMT with 15 workers, 15 helpers and one communication server per node, and use from 2 to 128 nodes.

Figure 2a shows the weak scaling of a simple queue-based BFS (one memory access per edge) on GMT, measured in million of traversed edges per second (MTeps). The graphs used for these experiments are randomly generated, with 1 million of vertices for every node added. Each vertex has at most 4000 edges connecting to random vertices in the graph. Therefore, the larger graph used with 128 nodes has 128 million vertices and 258 billion edges for a memory footprint of ≈ 2 TBytes. GMT's performance scales almost linearly while the size of the of the graph increases. Figure 2b shows the strong scaling of BFS on GMT, compared to the equivalent queue-based implementations for UPC, the Cray XMT. The UPC implementation runs on Olympus. Given the memory limitations of the other systems, for strong scaling experiments we used a random graph composed of 10 million vertices and 2.5 billion edges. We see that BFS on GMT outperforms the other implementations. However, because the graph is relatively small, and GMT needs 2 million tasks to fully utilize a system with 128 nodes and 15 workers, its performance starts to decrease above 64 nodes.



(a) Weak scaling



(b) Strong scaling

Figure 2: BFS results on GMT

5. 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 fork/join parallelism. It provides a simple interface for designing applications with large, irregular data structures, without requiring data partitioning. It allows expressing and extracting the large amounts of fine grained, dynamic parallelism present in irregular applications through simple parallel loop constructs. GMT's architecture employs specialized threads to realize its functionality. The library 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.

6. REFERENCES

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

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