UPC Queues for Scalable Graph Traversals: Design and Evaluation on InfiniBand Clusters

Jithin Jose Sreeram Potluri Miao Luo Sayantan Sur Dhabaleswar K. (DK) Panda

Department of Computer Science and Engineering, The Ohio State University

{jose, potluri, luom, surs, panda}@cse.ohio-state.edu

1. Abstract

PGAS languages like UPC are growing in popularity because of their ability to provide shared memory programming model over distributed memory machines. While this abstraction provides better programmability, some of the applications require mutual exclusion when operating on shared data. Locks are a common way to achieve mutual exclusion in shared memory algorithms. However, they impose a huge performance penalty on distributed memory machines and have been shown to be one of the major scaling bottlenecks. Simplistic approaches to eliminate locks by replicating resources is inherently non-scalable due to high memory cost.

In this paper, we introduce a UPC library that provides the abstraction of Queues. Our UPC library is tightly integrated with underlying UPC Runtime and utilizes Active Messages. The implementation of Active Messages provides implicit mutual exclusion that we exploit to design Queues. We present the design and implementation of Queues in UPC and compare their performance with that of existing mechanisms for operating on shared data with mutual exclusion. We evaluate our approach by re-designing two popular graph benchmarks: Graph500 and Unbalanced Tree Search (UTS), using Queues in UPC. Experimental results indicate that queue-based implementation for Graph500 outperforms the replication-based implementation by around 44% and 30% for 512 and 1,024 UPC-threads, respectively. Performance improvements of queue-based version of Unbalanced Tree Search (UTS) benchmark over the current version are about 14% and 10% for similar scale runs, respectively. Our work is based on the Berkeley UPC Runtime and the Unified Communication Runtime (UCR) for UPC and MPI, developed at The Ohio State University.

2. Introduction

Partitioned Global Address Space (PGAS) languages improve ease of programming by providing a shared memory abstraction on distributed memory machines [10]. They also provide control of data layout and work distribution which allows applications developers to take advantage of locality. Unified Parallel C (UPC) [30], a dialect of C (ISO C99), is one of the most popular languages in the PGAS family. Several flavors of UPC are available along with implementations for the compiler and runtime on a variety of architectures including commodity clusters and leadership class machines such as the Blue Gene/P [4, 20, 28]. The Berkeley UPC implementation [20] is a widely-used open-source implementation that has support on several high-performance interconnects such as Myrinet, InfiniBand and Quadrics through the portable GASNet runtime [13].

Graphs are one of the most ubiquitous models in analytical workloads. They are powerful representations of many types of relations and process dynamics and are used in a variety of scientific and engineering fields like cyber security, medical informatics, social networks, symbolic networks, financial applications, etc. Basic graph algorithms such as breadth-first search, depth-first search, shortest path, minimum spanning tree, etc. are key components in many modern real-life applications [11], [25], [32], [15]. For these reasons, optimizing graph algorithms is of high importance. Benchmarks that help analyze behavior of clusters in the context of graph algorithms are also gaining popularity. The Graph 500 [2], announced at International Supercomputing Conference (ISC'10), provides a set of graph benchmarks for evaluating and ranking supercomputers across the world. Another popular graph benchmark is the Unbalanced Tree Search (UTS) [26]. These benchmarks represent applications with data intensive and irregular communication patterns. The benchmark results provide useful insight into performance of various systems for similar workloads.

2.1 Motivation

Distributed graph algorithms require communication between peers. In UPC, this is expressed by threads modifying globally shared data structures It is common for multiple threads to simultaneously exchange data with a single thread. Mutual exclusion is required in such scenarios, to ensure correctness. Schemes based on locks are commonly used to achieve this. Locks on shared-memory systems are infamous for contention and come with a considerable overhead on distributed-memory architectures, therefore, they are not scalable. Another approach to this is to replicate resources and each of the remote threads operate on one of these resources. This scheme achieves mutual exclusion between sending threads but, leads to increased memory consumption and impacts performance due to the overhead of polling replicated resources. The impact of polling becomes significant as the system scale increases. To quantitatively characterize the scalability issues of operating on shared UPC data structures using existing mechanisms, we performed the following experiment. In our experiment, every thread sends a fixed amount of data to thread0, and thread0 sends back the data to the sender thread. A naive way to implement this in UPC is to use locks, by having a receive buffer at thread0 and preventing simultaneous access through a lock. As we can see from Table 1, this scheme does not scale. Another method to implement this benchmark is to use notification array with resource replication. In this scheme, separate receive buffers are kept for each remote thread. Each of the remote threads puts data into the corresponding buffer.

Fifth Conference on Partitioned Global Address Space Programming Model October-2011, Houston, TX.

Copyright © 2011 ACM [to be supplied]. . . $10.00

No. of Threads	**4**	**16**	**64**	**256**
UPC Locks	24	135	610	2610
Notification Array (Replication)	6	31	138	610

Table 1. Average send-recv time using Locks and Resource Replication schemes for 128 bytes data (time in usec)

Then it sets a specific value in the notification array to indicate that it has put data. Thread0 polls notification array to identify which all threads have put data. Even though the latency decreases a little, it requires more memory (O(N) per process).

Queues provide a good abstraction for managing producer-consumer relationships. It can be implemented efficiently in a contention-free manner using active messages, while having a small memory footprint and minimal polling overhead. This motivates the work in this paper to introduce the concept of queues with implicit mutual exclusion in UPC and demonstrate how they can be used to address mutual exclusion overheads in graph benchmarks.

2.2 Contributions

This paper makes several important contributions. They are listed as follows:

1. We introduce Queues with implicit mutual exclusion in UPC and present their design and implementation in UPC runtime.
2. We demonstrate how data intensive irregular applications in UPC can be redesigned using Queues for better performance. We use Graph500 and Unbalanced Tree Search (UTS) benchmarks as examples.
3. We provide detailed performance evaluation using a set of micro-benchmarks and the aforementioned application benchmarks.

Graph500 benchmark using Queues outperforms the naive implementation, based on standard UPC primitives, by around 44% and 30% for 512 and 1,024 UPC-thread runs, respectively. Performance improvements for queue variation of UTS over the existing version are about 14% and 10% for similar scale runs, respectively. Our work is based on the Berkeley UPC Runtime and the Unified Communication Runtime (UCR), developed at The Ohio State University. It throws light to the possibility of tremendous performance improvement and ease of programmability using the Queue concept in PGAS languages.

The rest of the paper is organized as follows. Section 3 describes the existing mechanisms for implementing queues and related work in the field of optimizing graph algorithms for PGAS models, and contrast our work. In Section 4, we describe background material for this work. In Section 5, we state the design requirements for queues in UPC, present queue operations and provide details of design and implementation of queues in UPC runtime. In Section 6, we demonstrate that data intensive irregular applications can be redesigned with queues using two popular graph benchmarks: Graph500 and UTS. Then, in Section 7, we provide a detailed evaluation of our implementation of queues using micro-benchmarks and the aforementioned graph benchmarks. We conclude in Section 8 and discuss future directions

3. Related Work

In this Section, we describe associated background work that is relevant to this paper. We also discuss about other recent and related work in this area.

PGAS Graph Optimizations: Irregular graph algorithms with data intensive workloads and random access pattern are known to be notoriously hard to implement and optimize for high performance on distributed-memory systems. Even though shared-memory algorithms can easily be mapped onto distributed-memory systems using PGAS languages, these might not result in high performance due to irregular communication pattern and lack of locality. G. Cong et. al. investigated this in detail and proposed a set of optimization techniques for improving access locality [11], which helped in improving communication performance as well. They also proposed optimization techniques such as compaction, offloading, circular orchestration, etc. J. Zhang et. al analyze the UPC implementation of Barnes Hut algorithm [5] in [32] and apply successive optimization techniques to tremendously improve performance. Most of the optimizations they considered were in one way or another related to improving the locality. In our work, we propose Queues in UPC and show how the graph algorithms can be implemented efficiently using this. The optimization techniques proposed in [11] and [32], and our work are orthogonal. The optimization techniques proposed in these works can be applied along with the use of UPC Queues.

Message Passing Runtimes and Active Message Libraries: Supercomputer systems designed specifically for data intensive and irregular applications are available [1]. These systems are massively multithreaded and target parallel applications that are dynamically changing and require random access to shared memory. High performance message passing runtimes [19] for specific systems platforms have been proposed. Such runtimes are tightly dependent on the underlying machine architecture and special interface classes need to be implemented for other target architectures. Jeremiah et. al proposed Active message library [31]. Such user libraries implemented over MPI introduce overheads from MPI stack, such as tag matching, sender matching, etc. On top of this, extra data bytes need to be transferred for MPI headers. Usability is another important aspect. In case of user level active messaging libraries, specific message handlers need to be implemented. UPC Queues provide a simple and easy to use interface. Since the UPC Queues are implemented over GASNet, no additional overheads are imposed. Design and implementation details of UPC queues are mentioned in detail in Section 5.

4. Background

In this section, we provide a brief overview about the UPC GASNet system architecture, InfiniBand communication system and Unified Communication Runtime (UCR). We also provide a brief overview about the two graph benchmarks, Graph500 and UTS.

4.1 UPC and GASNet Communication System

Unified Parallel C (UPC) [30] is an emerging parallel programming language that aims to increase programmer productivity and application performance by introducing parallel programming and remote memory access constructs in the language. UPC is based on the Partitioned Global Address Space (PGAS) programming model. The PGAS programming model allows programmers to view a distributed memory supercomputer as a global address space, that may be partitioned to improve performance. There are several other PGAS programming languages, namely X10 [9], Chapel [8] and HPF [21], along with Global Address Space libraries such as Global Arrays [14].

The runtime implementations of UPC have been demonstrated to be scalable and provide very good performance to end applications through fine-grained remote memory accesses [7] and improved communication overlap [24]. In particular, the Blue Gene implementation of UPC, developed by IBM, has been demonstrated to be highly scalable [6]. In this paper, we focus on the InfiniBand implementation of UPC through the popular GASNet communication library [13]. GASNet is a language-independent, low-level networking layer that provides network-independent, high-performance communication primitives tailored for implementing parallel global address space SPMD languages such as UPC, Titanium, and Co-Array Fortran.

UPC Runtime makes use of GASNet interface for remote memory updates and accesses. This interface consists of Core APIs and Extended APIs [13]. The core API interface is a narrow interface based on the Active Message paradigm. Extended APIs provide a rich, expressive and flexible interface that provides medium and high-level operations on remote memory and collective operations. GASNet supports different network conduits, viz., ibv (OpenIB/OpenFabrics IB Verbs), udp (UDP), lapi (IBM LAPI) [16], mpi (MPI), etc. In [18], we proposed a new high performance conduit for InfiniBand networks. This work is described in detail in Section 4.3.

4.2 InfiniBand

InfiniBand [17] is an open industry standard switched fabric that is designed for interconnecting nodes in High End Computing clusters. It is a high-speed, general purpose I/O interconnect that is widely used by scientific computing centers world-wide. The recently released TOP500 rankings in June 2011 indicate that more than 41% of the computing systems use InfiniBand as their primary interconnect. One of the main features of InfiniBand is Remote Direct Memory Access (RDMA). This feature allows software to remotely read memory contents of another remote process without any software involvement at the remote side. This feature is very powerful and can be used to implement high-performance communication protocols. InfiniBand has started making in-roads into the commercial domain with the recent convergence around RDMA over Converged Enhanced Ethernet (RoCE) [29].

4.3 Unified Communication Runtime (UCR)

UCR provides a unified and high performance communication runtime that supports multiple programming models. This facilitates hybrid programming models without having the overheads of separate runtimes and their inter-operation. UCR was first proposed in [18], (previously called INCR) and introduced a new high performance InfiniBand conduit for GASNet, (GASNet-UCR), which supports MPI and UPC communications simultaneously. The project draws from the design of MVAPICH and MVAPICH2 [23] software stacks. UCR has been optimized with multiple endpoints design in order to enhance the performance of multi-threaded UPC runtime [22]. High performance computing runtime and distributed communication models share significant overlap. UCR provides a clean and simple interface for supporting different programming models and even data center applications such as Memcached.

4.4 Graph500

Graph500 Benchmark Specification [3] is proposed to direct design of a new set of benchmarks that can evaluate the scalability of supercomputing clusters in the context of data-intensive applications. Graph500 benchmark stresses hardware and runtime systems by forcing massive amounts of communication and synchronization thereby modeling more realistic application workloads. The Graph 500 was announced at International Super Computing, 2010 (ISC'10) and the first Graph500 ranking appeared at Super Computing, 2010 (SC'10). The ranking of systems based on Graph500 is released twice every year, in June and November. It consists of three comprehensive benchmarks to address application kernels: Search (Concurrent Search), Optimization (Single Source Shortest Path) and Edge Oriented (Maximal Independent Set). These represent business area data sets, such as cyber security, medical informatics, data enrichment, social networks, symbolic networks, etc. We focus on concurrent search benchmark in this paper, which basically does BFS traversal of the graph.

Concurrent search benchmark consists of three phases (termed as kernels in benchmark specification). The first is 'Graph Construction', which generates edge. The 'Kronecker Generator' algorithm is used in the reference implementation. From the edge list, graph is constructed in Compressed Sparse Row (CSR) format. Each UPC thread keeps the adjacency list information of vertices that it owns. Vertex ownership is defined in cyclic manner (vertex number modulo number of processes). The second kernel is the actual 'Breadth-First-Search'. In kernel2, 64 search keys are randomly sampled from the vertices in the graph. For each of these search keys, BFS traversals are made, one by one. The final kernel is the validation kernel, which ensures the correctness of BFS traversal.

Several versions of referenced benchmark (Concurrent Search) implementation are available on Graph500 List [2], including sequential, OpenMP, XMT, and MPI.

4.5 Unbalanced Tree Search (UTS)

The Unbalanced Tree Search (UTS) benchmark is designed to evaluate the performance and programmability of parallel applications that require dynamic load balancing [27]. It targets the problem of performing an exhaustive search on an implicitly defined tree: any node in the tree has enough information to construct its sub-tree from the description of its parent, using SHA-1 cryptographic hash function [12]. The tree is constructed on the fly during the search process. Load balancing issue arises because of the variation in the sizes of sub-trees generated at different nodes. Parallel implementation of the search requires continuous dynamic load balancing to keep all processors engaged in the search [25]. While this can be solved using work sharing or work stealing, the later has been shown to be more efficient. Processes/threads that become idle can steal nodes from others with minimum disturbance to the search process. UTS has been implemented using several popular parallel programming models: UPC, OpenMP, MPI, Shmem, Pthreads, Chapel and X10, etc. Earlier work has optimized the implementation in UPC to achieve up to 80% efficiency on 1,024 processors [25, 26]. It has been shown that UPC provides ease of expressing asynchronous work-stealing protocols compared to message passing models. We have used 'uts_upc_enhanced' (in UTS v1.1) as the base version for our work. This is one of the most optimized and enhanced implementations of UTS benchmark.

5. Design

In this section, we first discuss the design requirements for queues in UPC and explain how these can be satisfied in UPC Queues. Then we present the different queue operations followed by their design and implementation using active messages.

5.1 Design Requirements

We expect that a design for queues in UPC should satisfy the following requirements

1) **Programmability:** Ease of programming is an important reason for the growing popularity of PGAS languages in general and UPC in particular. Ensuring this is imperative for the acceptance of any new extensions to the UPC standard. Hence we consider this the first requirement while proposing queues in UPC. Queues can improve programmability compared to the alternatives like locks and resource replication. For example, work delegation to a peer thread which involves the sequence of lock() - memput() - unlock() calls in a lock-based design can be replaced by a single enqueue call in a queue-based application. Queues can avoid the complexity of

managing and polling multiple data structures for incoming work messages as is the case of a replication-based design.

2) **Scalability:** The fact that existing mechanisms for achieving mutual exclusion in UPC do not scale well is the key motivation factor behind our work. The design of queues should be centered around this requirement. It should avoid the contention seen with locks on one hand, while minimizing memory and polling overheads on the other. Active messages in GASNet provide a good option for implementing queues. As will be presented in the later sections, queues implemented over active messages can achieve very good scalability with a small memory footprint and minimum polling overhead.

3) **Low Latency:** It is necessary that queue operations have minimal overhead as compared to the lower level interface on which they are implemented. Work delegation requests, or control messages can be considered as one of the major use cases for UPC queues. It is imperative that these requests are delivered with low latency. As will be discussed in Section 5.3, queue operations map closely onto active messages, have minimum overheads and can achieve latencies close to that of active messages.

4) **Portability:** As UPC is used on a myriad of system architectures, it is important for its features to be portable without sacrificing performance. The design of queues should not be based on any single network (or conduit) or specific architecture, and should ensure portable performance. We implement queues over active messages provided by GASNet. GASNet can be configured to use any of the conduits, like IBV (for InfiniBand), UDP, SMP, MPI (GASNet implementation over MPI semantics), etc. Thus, queues inherently gain this portability. Our evaluation in Section 7 shows that queues can achieve similar benefits in performance over different conduits.

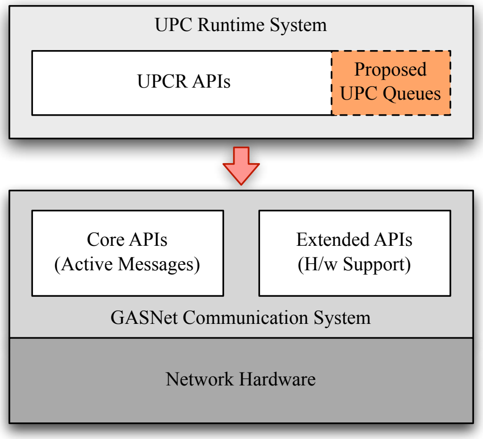

UPC Runtime System
UPCR APIs
Proposed UPC Queues
Core APIs (Active Messages)
Extended APIs (H/w Support)
GASNet Communication System
Network Hardware

Figure 1. Implementation of Queues in UPC Runtime

5.2 UPC Queue Operations

In this section, we present an overview of the operations that can be performed on the proposed Queues in UPC. The work in this paper focuses more on the performance and productivity aspects of queues in UPC, rather than on the syntax of queue APIs. Queue access APIs can be made in sync with UPC language constructs using efficient compiler translation techniques. We emphasize more on the concept of queues in UPC and demonstrate how applications can be implemented efficiently using it. The current implementation of queues is done in the UPC runtime layer and it supports five basic operations. Each of these are explained in detail below.

1) `upc_queue_create`: This operation creates an instance of UPC Queue and returns a handle. All subsequent queue operation calls use this handle to identify the queue instance. This is designed as a collective call. UPC queue supports coalescing of queue elements, which avoids the communication cost for each of the enqueue operation. This feature is optional and can be enabled or disabled using specific flags in `upc_queue_create`.

2) `upc_queue_enqueue`: Queue items can be enqueued using this function. If coalescing is enabled, queue item is buffered locally, until the local bucket for the target is full or until an explicit flush call is made. Otherwise, it is sent immediately. Queue item can be any data, and the size of the data is indicated as an input parameter to this operation.

3) `upc_queue_dequeue`: This function call is used to dequeue items from the queue. It can operate in two modes, blocking and non-blocking. In case of blocking mode and if the queue is empty, the function call blocks until an item is put into the queue or until a specified timeout. If queue is not empty, the call returns immediately, and the queue item gets dequeued. In case of non-blocking mode, function call returns immediately whether or not queue is empty. In-out argument indicate the size of queue item, which gets dequeued. If the queue is empty, this argument is set as zero when the function call returns.

4) `upc_queue_flush`: This function is needed only in coalescing mode. As indicated above, in coalescing mode, queue items are buffered until the bucket is full. `upc_queue_flush` can be used to flush out any such buffered queue items. This is designed as a non-collective call, for better programmability.

5) `upc_queue_destroy`: Queue can be destroyed using this function call. Any resources allocated for supporting queue operations are released in this call. This is designed as a collective call.

5.3 Design and Implementation

The proposed 'UPC Queues' are implemented in UPC Runtime layer. UPC Runtime layer employs GASNet interface for remote memory updates and shared memory accesses. GASNet provides active messaging interface as well as direct remote memory access interface. Implementing queues over direct remote memory accesses will again require explicit locking mechanisms which have inherent performance and scalability constraints. We considered GASNet active messages for implementing UPC Queues, because active message semantics match very well with the UPC Queue implementation requirements. They provide implicit mutual exclusion when executing at a given thread.

Active messages are similar to normal messages, but it invokes a handler function at the receiver side. The handler function is selected based on the handler id, which the active message carries along with. It also carries arguments with which the handler shall be executed. GASNet Active Message interface consists of three APIs, namely, `gasnet_AMRequestShort, gasnet_AMRequestMedium and gasnet_AMRequestLong` [13]. This classification is based on message size. `gasnet_AMRequestShort` active message carries only handler id and arguments without any data payload, whereas `gasnet_AMRequestMedium` carries payload along with the handler id and arguments. `gasnet_AMRequestLong` is designed for large payloads. It puts the payload directly in the target location at the receiver side and then, the message handler is executed. The target location needs to be known while sending the message.

We chose medium active messages to implement queue operations. Small active messages do not carry payloads; and in case of long active messages, the target location needs to be known while sending the message. These do not satisfy the requirements of enqueue and dequeue operations that have a payload and operate on a queue handle rather than specific addresses. Medium active messages address these requirements well.

The enqueue operation works as follows: Enqueue operations invoked from UPC application layer is translated into GASNet medium messages at runtime layer. Queue item is set as the payload, and the handler identifier for queue operation is set as the handler id. When an active message arrives at the receiver side, it is buffered in the UPC runtime layer. Reception of active message and the buffering is transparent to the UPC application layer. The queue item is given to the application layer only when it invokes dequeue operation.

UPC Queues provide coalescing of queue items. Coalescing avoids the communication costs for each enqueue operation. Multiple queue items destined for a remote thread are aggregated and are sent out as a single active message. In order to support coalescing, separate buckets are kept for each of the remote UPC threads. As only one bucket buffer is required for each remote thread, the memory requirement is not substantial. These buckets are created during the queue creation time, based on the coalescing size specified. During an enqueue operation, the queue item is put into the bucket designated for the destination thread. The data is sent out, when the bucket is full, or when a flush operation is called by the UPC application. If enqueue operation is invoked without coalescing (immediate enqueue), the queue item is sent out immediately.

UPC Queue enqueue/dequeue operation is explained in detail in Figure 2. The diagram depicts the scenario when coalescing is enabled. When UPC application invokes `upc_queue_enqueue`, the queue item is buffered into the respective bucket for the target. This is indicated as (1) in the figure. When the bucket is full or, when an explicit `upc_queue_flush` is invoked, the bucket is sent over active message to the target UPC thread (2). Active message handler at the target side enqueues this into the queue (3). When the application layer at target side invokes `upc_queue_dequeue`, the queue item is dequeued from the queue and is given to the application (4).

The maximum payload that an active message can carry differs for different GASNet network conduits. It is determined based on performance tuning and conduit level optimizations. If the queue item size or the coalesced size is greater than the maximum payload it can carry, it is sent out in chunks. These chunks are reassembled at the receiver side, during the active message handling.

User level active message libraries like AM++ [31] have been proposed. Queue operations can also be implemented over such active message libraries. In this approach, the translation of enqueue/dequeue operations into active messages, handling of active messages, etc. need to be handled in the UPC application layer. This stands against our design consideration of programmability. Another aspect is performance. Active message libraries implemented over MPI libraries impose software overheads from MPI stack, whereas in our approach, the active messages are implemented directly over the GASNet, which does not impose additional overheads.

In this work, we focus on the concept of queues in UPC and how this can used to implement applications with irregular access patterns, in an efficient manner. We implemented queues in UPC runtime layer to demonstrate the usability and performance improvements. Using efficient compiler translation techniques, the queue concept can be implemented in sync with UPC language constructs.

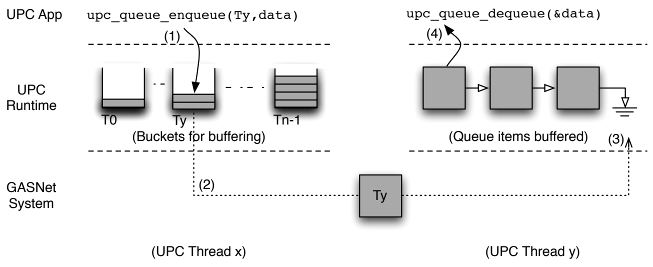


Figure 2. Enqueue/Dequeue operation in UPC Queue

6. Redesigning Applications using UPC Queues

In this section, we demonstrate how queues can be used to design UPC applications for better performance and scalability. We do this by re-designing two popular graph benchmarks: Graph500 [3] and Unbalanced Tree Search (UTS) [27]. Both these benchmarks exhibit irregular communication pattern but represent two different use cases for queues. For each of these benchmarks, we first present an overview and a description of their current implementations. Then, we discuss their implementation using queues.

6.1 Graph500

In this section, we first describe the communication characteristics of Graph500 concurrent search benchmark and then describe the reference MPI implementation. We introduce the UPC version of concurrent benchmark and then present the version using queues.

Concurrent search benchmark does Breadth First Search (BFS) traversal on a given graph. The graph is stored in a compressed row format and is distributed across the processes/threads, each process/thread owning a subset of vertices. Process owning the root vertex starts the traversal by exploring the neighbors of the root. If the newly discovered vertices are local, they are marked for traversal. If a discovered vertex is not local, the owner process is notified which marks the vertex for traversal. The communication in this benchmark involves a large number of small messages, exchanged in an irregular manner depending on the connectivity of the graph.

Reference Implementation using MPI: In the reference implementation of Graph500 using MPI, concurrent search is implemented using two queues, ‘new_queue’ and ‘old_queue’ and the traversal happens in a level-based fashion. At each level, vertices from old_queue are dequeued and their adjacent vertices are examined. If the newly discovered vertex is local, it is added into new_queue. Otherwise, notification is sent to the owner process in the form of a MPI message. On receiving a message, the process checks if this vertex has been already visited. If this vertex is unvisited, this is enqueued to its new_queue. Or else, it is ignored. This benchmark uses message coalescing for optimizing the communication performance. Notification of newly discovered vertices destined to a single remote process are aggregated locally, and are sent in one MPI_Send.

End of level is identified when all the vertices in old_queue have been processed. All processes synchronize at the end of each level to check to see if there are any new nodes to be visited. This is done by performing an MPI_Allreduce on the number of elements in new_queue. If the new_queue is empty in all the processes, the traversal is considered as complete. Otherwise, new_queue and old_queue are swapped and the traversal continues to the next step. A detailed description of the benchmark can be found in [3].

UPC Version Using Notification Array: Since UPC reference implementation is not available in Graph500 benchmark suite, we implemented the UPC version based on the specification. To the best of our knowledge, this is the first UPC version of Graph500 benchmark. In this benchmark, we implement the new_queue as a shared array so that any UPC-thread can enqueue vertices remotely. We used coalescing of vertices, similar to that of MPI reference implementation. Coalescing is one of the the key methods for improving communication efficiency in PGAS models [11], [32]; because each shared array access will involve considerable communication cost when it is on a remote node. We used a coalescing size of 4KB in our implementation same as that of MPI. Other optimization techniques mentioned in [11] such as 'circular' and 'local copy' are also being used. In this implementation, the enqueues to remote UPC thread is done using `upc_memput`. It uses a separate shared array for notification of incoming enqueue. We refer to this version of benchmark as 'UPC (notification array)' in the rest of the paper.

UPC Version using Queues: We enhanced Graph500 by using the proposed UPC queues to implement new_queue. Since queues provide in-built coalescing, we used this feature. We used the same bucket size as that of the 'UPC (notification array)' version of benchmark. Vertices are enqueued to remote UPC-threads using `upc_queue_enqueue` call, and receiver UPC-thread dequeues it using `upc_queue_dequeue` (used in non-blocking mode). At the end of each level, `upc_queue_flush` calls are made to flush out any pending enqueue operations. This version reduces the complexity of the application as it frees the developer from handling explicit notification and from designing optimization techniques like coalescing. This version is indicated as 'UPC (queue)' in the experimental results section.

6.2 UTS

In this section, we demonstrate yet another use case for queues. We use queues for sending and receiving control messages in UTS benchmark. We first describe the existing UPC version that was introduced in Section 4.5. Then, we show how it can be enhanced using queues.

The UTS benchmark performs an exhaustive search on an implicitly defined tree. As tree construction happens during the traversal, it is common for the tree to grow much faster at some processes/threads than at others. This demands dynamic load balancing to keep all the processors busy. The processes that become idle send out work requests to their peers in a cyclic order. These requests are serviced by threads which have enough work to delegate. As the distribution of work is irregular, the exchange pattern of work requests and responses ends up being quite random. The size of these packets is small, as they contain control and address information and not the actual data.

Existing UPC Version: We present the existing UPC version of UTS benchmark here. This implementation has been extensively optimized using techniques presented in [25–27], and is called 'uts_upc_enhanced' in the UTS benchmark suite. In this implementation, thread0 starts the search from the root vertex. As traversal progresses, the tree is constructed on the fly and the newly identified vertices are added to a shared stack. A thread that is idle sends work requests to other threads starting with the thread whose id is one greater than its own. The work request is made by putting its own id into a lock-protected shared variable in the remote thread's memory. Working threads check the shared variable periodically for work requests. If there is enough work to share, they delegate work to the requester thread by modifying another shared variable in that thread's memory. If a work request is denied, the requester thread will continue requesting other threads. If it identifies that some thread is willing to share work, then the vertices (work) are fetched using a `upc_memget` operation. The benchmark exits when the graph traversal is done. A detailed description about this benchmark operation is provided in [25]. This benchmark is denoted as 'UPC (base version)' in the performance evaluation section.

UPC Version with Proposed Queues: We enhanced the UTS benchmark to use queues for control messages. In this version, UPC threads make use of queues for work requests and replies, instead of using lock protected shared variables. When an idle thread wants to request work, it does so by enqueueing a work request to a busy thread. Busy threads check queue periodically for incoming work requests. Willingness or unwillingness to share work is indicated to the requester thread by enqueueing a work response packet to the requester thread. Actual work is transferred using `upc_memget` operation, as in the case of the 'uts_upc_enhanced' version. This version of the benchmark is indicated as 'UPC (queue)' in the performance results. All other parameters and configurations, such as polling frequency, work share size, etc are exactly the same as that of the 'uts_upc_enhanced' variant.

7. Experimental Results

Here, we compare the performance of the proposed UPC queues with that queue operations implemented using existing primitives in UPC. We considered UPC queue implementation using resource replication with shared notification arrays and using UPC lock primitives. We evaluate these using representative benchmarks and then discuss how UPC applications can benefit using queue approach. We start with describing the experiment platform followed by micro-benchmark and application evaluations.

7.1 Experimental Platform

We used an Intel Westmere cluster for our experiments. This cluster consists of 144 compute nodes with Intel Xeon Dual quad-core processor nodes, operating at 2.67 GHz. Each node has 12GB of memory and is equipped with MT26428 QDR ConnectX HCAs (36 Gbps data rate) with PCI-Ex Gen2 interfaces. The nodes are interconnected using 171-port Mellanox QDR switch. The operating system used is Red Hat Enterprise Linux Server release 5.4 (Tikanga), with kernel version 2.6.18-164.el5 and OpenFabrics version 1.5.1.

We used Berkeley UPC version 2.12.2 [20] for micro-benchmark and application performance evaluation. This is the latest available Berkeley UPC version. In all the experiments, we used single UPC thread per process configuration mode. We used GASNet-UCR conduit for benchmark evaluation and both GASNet-UCR and GASNet-IBV conduits for application performance evaluation. GASNet-UCR is the GASNet InfiniBand conduit proposed by OSU in [18] and GASNet-IBV is the native GASNet InfiniBand conduit, implemented over InfiniBand verbs API's. The research in [18] states that GASNet-UCR performs identical to that of the GASNet-IBV conduit in UPC level evaluations. The MPI library used in micro-benchmark evaluation is MVAPICH Library [23].

7.2 Micro-benchmark Performance

In this section we compare the performance of enqueue-dequeue operations in the proposed UPC Queues with that of implementations using existing alternatives for synchronization. We first give an overview of the different alternatives considered. Then we provide details about the benchmark and finally present the experimental results.

A common way to implement queue operations in UPC is using shared arrays and UPC locks. Even though the implementation is simple and it matches with the shared-memory programming style, such a design will not scale because of the lock contention. In the case of distributed-memory architectures, it involves a considerable communication cost. Another way to implement queues is by keeping dedicated regions in shared array for each of the remote threads and using shared notification arrays. The enqueue operation is notified using the notification array, which is also a shared array. A UPC thread polls the notification array for checking if there are any enqueue operations made by remote threads. The part of notification array that each thread polls, is local to itself so that no network operation is involved during polling. We included this design in micro-benchmark evaluation. This design is denoted as 'UPC (notification array)' in the graphs.

Another way for implementing the queues in UPC is using MPI Send/Receive semantics (a hybrid UPC+MPI model). Enqueues and dequeues can be translated as `MPI_ISend` and `MPI_IRecv` calls, respectively. Even though this approach looks simple, it imposes overhead from MPI software stack. This is denoted as 'MPI' in the graphs.

Benchmark: Our benchmark aims to characterize performance and scalability behavior of the different schemes discussed above. In this benchmark, every UPC thread enqueues to thread0, and thread0 on dequeueing this element, enqueues it back to the remote thread. This pattern is characteristic of real world data intensive applications where multiple threads can simultaneously communicate with a single thread. As the number of threads increases, the contention at thread0 will help evaluate performance of the different schemes in these scenarios. Each thread enqueues 1,000 elements. We measure the average time for a single enqueue-dequeue operation.

Latency: Figure 3 compares the average time of an enqueue-dequeue operation for different queue item sizes. We conducted this experiment for different number of UPC threads. Results indicate that UPC (queue) implementation performs better than other designs, for all the message sizes. In the experiment with 128 UPC threads, enqueue-dequeue operation using the proposed queues in UPC achieves 94% lower latency when compared to an implementation using locks, for a payload of 128 Bytes. We see 74% lower latency when compared to an implementation using replication.

Scalability: Figure 4 presents the performance of queues from a scalability perspective. It shows the performance cost of enqueue-dequeue operation for a specific queue item size (128 byte), for varying number of UPC-threads. The number of UPC threads is plotted on X-axis and the time for enqueue-dequeue operation is plotted on Y-axis. Results indicate that proposed UPC queue implementation scales the best.

Micro-benchmark evaluations clearly state that UPC queues do perform much better as compared to other queue implementations. It also highlights that this design does not introduce any overhead even as the number of UPC threads increases.

7.3 Graph500 Benchmark Performance

We used Graph500 version 1.2 for our experimental evaluation. We ran the benchmark for an input graph with 16 million vertices and 256 million edges, for varying number of system sizes, 64, 128, 256, 512 and 1,024 UPC-threads. We conducted this experiment with both the high performance InfiniBand GASNet conduits for UPC, GASNet-IBV and GASNet-UCR. Results of these conduits are indicated with '[ibv]' and '[ucr]' postfixes, respectively.

Performance results of Graph500 application are presented in Figure 5(a). Results show tremendous performance improvement for the queues version, as compared to the base version. This is because of the fact that, in queues design, we eliminate polling cost and the cost for extra message for notification. Since Graph500 is a data intensive benchmark, these costs are quite visible. Results

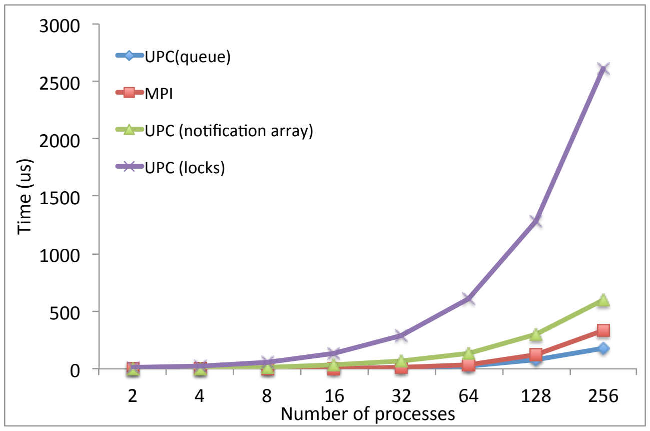


Figure 4. Performance comparison of enqueue-dequeue operation in different implementations of queues, for a 128 byte queue item, on varying system sizes

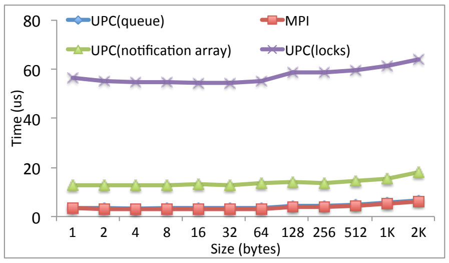
indicate that for a 1,024 UPC-thread run, UPC queues design obtains 30% improvement over the notification array design for both GASNet-IBV and GASNet-UCR conduits. For a 512 UPC-thread run, we observe about 44% and 48% improvement for GASNet-IBV and GASNet-UCR conduits, respectively. We observe that the BFS time increases with the number of threads beyond 256 threads. This is because of the strong scaling used in this experiment.

7.4 Unbalanced Tree Search Benchmark Performance

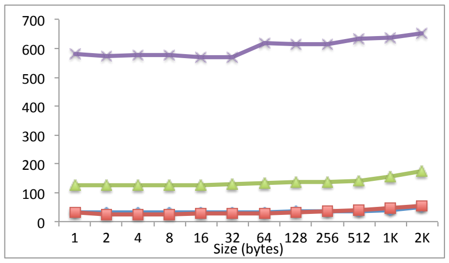
Figure 5(b) shows the performance comparison of UTS benchmark (denoted as 'UPC (base version)') and our version using queues. We used UTS Benchmark suite v1.1 for our evaluation. The benchmark was run with an input graph of 270 billion nodes (indicated as T1WL in the benchmark specification). We ran this experiment for different number of UPC-threads - 64, 128, 256, 512 and 1,024. Each of these were run using the InfiniBand GASNet conduits, GASNet-IBV and GASNet-UCR. We observed performance improvement for both these conduits with the queue based design. For a 512 UPC-thread run, queue version performs better than the 'uts_upc_enhanced' version by around 10% for both the conduits. For 1024 UPC-thread run, the performance gain is around 14% for GASNet-UCR conduit and 12% for GASNet-IBV conduit. The main reason for performance improvement is that we avoid the lock contention cost. On top of that, every lock or unlock operation results in a network communication. So for each request, there will be two network communication, whereas queue design requires only one network communication, which is the actual enqueue operation. We did not see much performance improvement for UTS benchmark as compared to the Graph500. This is because UTS benchmark is highly optimized. The lock access pattern in UTS is such that every UPC thread first tries to acquire the neighboring thread's lock. This reduces contention among threads. We observed effects of strong scaling beyond 128 threads, similar to those observed with the Graph500 benchmark.

8. Conclusion

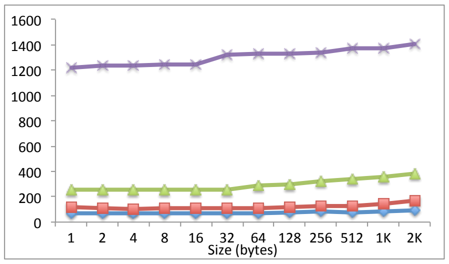
In this work, we introduce Queues in UPC to address contention and polling overheads in data intensive and irregular applications. We present the design and implementation of UPC Queues. We compare the performance of queues with that of alternative mechanisms currently available in UPC using micro-benchmark evalua-



(a) 8 processes

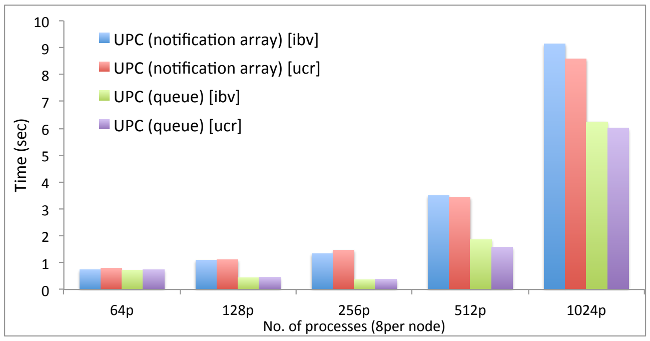


(b) 64 processes

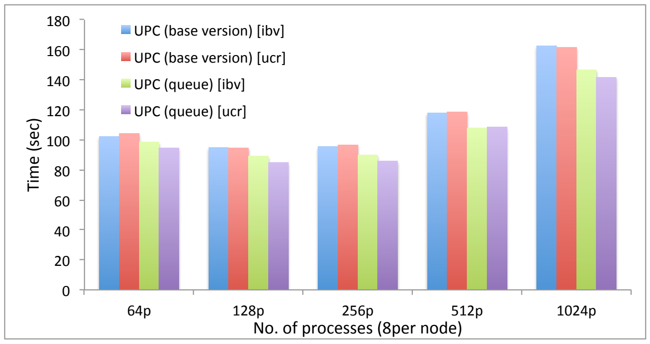


(c) 128 processes

Figure 3. Performance comparison of enqueue-dequeue operation in different implementations of queues



(a) Graph-500



(b) Unbalanced Tree Search (UTS)

Figure 5. Application Performance

tion. Finally, we demonstrate how the use of queues in data intensive applications by modifying two upcoming Graph benchmarks: Graph500 and UTS. Experimental results indicate that queue version for Graph500 outperforms the naive implementation by around 44% and 30% for 512 and 1024 UPC-thread runs, respectively. Performance improvements of queue variation of Unbalanced Tree Search (UTS) benchmark over the current version are about 14% and 10% for similar scale runs, respectively.

In this paper we emphasize on the Queue concept and demonstrate how data intensive and irregular applications can be redesigned using Queues. We would like to continue our work and propose extensions to UPC Language constructs for Queue operations, by making use of UPC compiler translations.

9. Acknowledgments

This research is supported in part by U.S. Department of Energy grants #DE-FC02-06ER25749 and #DE-FC02-06ER25755; National Science Foundation grants #CCF-0833169, #CCF-0916302, #OCI-0926691 and #CCF-0937842; grants from Intel, Mellanox, Cisco, QLogic, and Sun Microsystems; Equipment donations from Intel, Mellanox, AMD, Appro, Chelsio, Dell, Microway, QLogic, and Sun Microsystems.

We thank Dr. Xavier Besseron, Dr. Jerome Vienne and Dr. Hao Wang for their valuable support and feedback for this work.

References

[1] Cray XMT Architecture. `http://www.cray.com/products/XMT.aspx`.

[2] The Graph500 List. `http://www.graph500.org`, .

[3] Graph500 Specification. `http://www.graph500.org/specifications.html`, .

[4] GNU Unified Parallel C . `http://gcc.gnu.org/projects/gupc.html`.

[5] J. Barnes and P. Hut. A hierarchical O(NlogN) force-calculation algorithm. Nature 324 (1986), 446449.

[6] C. Barton, C. Casçaval, G. Almási, Y. Zheng, M. Farreras, S. Chatterjee, and J. N. Amaral. Shared memory programming for large scale machines. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 108–117, New York, NY, USA, 2006. ACM. ISBN 1-59593-320-4. doi: http://doi.acm.org/10.1145/1133981.1133995.

[7] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing Bandwidth Limited Problems Using One-Sided Communication and Overlap. In *Int'l Symposium on Parallel and Distributed Computing (IPDPS)*, 2006.

[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. ISSN 1094-3420. doi: http://dx.doi.org/10.1177/1094342007078442.

[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: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi: http://doi.acm.org/10.1145/1094811.1094852.

[10] 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*, PPoPP '05, 2005.

[11] G. Cong, G. Almasi, and V. Saraswat. Fast PGAS Implementation of Distributed Graph Algorithms. In *Super Computing (SC)*, 2010.

[12] D. Eastlake and J. P. US secure hash algorithm 1 (SHA-1). In *RFC 3174, Internet Engineering Task Force*, 2001.

[13] Editor: Dan Bonachea. GASNet specification v1.1. Technical Report UCB/CSD-02-1207, U. C. Berkeley, 2008.

[14] Environmental Molecular Sciences Laboratory and Pacific Northwest National Laboratory. The GA Toolkit. `http://www.emsl.pnl.gov/docs/global/`.

[15] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun. Accelerating CUDA Graph Algorithms at Maximum Warp. In *16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2011.

[16] IBM. Message Passing Interface and Low-Level Application Programming Interface (LAPI). `http://www-03.ibm.com/systems/software/parallel/index.html`.

[17] InfiniBand Trade Association. InfiniBand Industry Standard Specification. `http://www.infinibandta.org/`.

[18] J. Jose, M. Luo, S. Sur, and D. K. Panda. Unifying UPC and MPI Runtimes: Experience with MVAPICH. In *Fourth Conference on Partitioned Global Address Space Programming Model (PGAS)*, Oct 2010.

[19] S. Kumar, G. Dozsa, G. Almasi, D. Chen, M. E. Giampapa, P. Heidelberger, M. Blocksome, A. Faraj, J. Parker, J. Ratterman, B. Smith, and C. Archer. The Deep Computing Messaging Framework: Generalized Scalable Message Passing on the Blue Gene/P Supercomputer. In *Proceedings of the 22nd annual international conference on Supercomputing (ICS)*, 2008.

[20] Lawrence Berkeley National Laboratory and University of California at Berkeley. Berkeley UPC - Unified Parallel C. `http://upc.lbl.gov/`.

[21] D. Loveman. High performance Fortran. *Parallel Distributed Technology: Systems Applications, IEEE*, 1(1):25 –42, feb. 1993. ISSN 1063-6552. doi: 10.1109/88.219857.

[22] M. Luo, J. Jose, S. Sur, and D. K. Panda. Multi-threaded UPC Runtime with Network Endpoints: Design Alternatives and Evaluation on InniBand Clusters. In *18th Annual International Conference on High Performance Computing (HiPC)*, 2011.

[23] Network-Based Computing Laboratory. MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE. `http://mvapich.cse.ohio-state.edu/`.

[24] R. Nishtala, P. Hargrove, D. Bonachea, and K. Yelick. Scaling Communication-Intensive Applications on BlueGene/P Using One-Sided Communication and Overlap. In *Int'l Symposium on Parallel and Distributed Computing (IPDPS)*, 2009.

[25] S. Olivier and J. Prins. Scalable Dynamic Load Balancing Using UPC. In *Proceedings of 37th International Conference on Parallel Processing (ICPP)*, 2008.

[26] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C. Tseng. UTS: An Unbalanced Tree Search Benchmark. In *Proceedings of 19th Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2006.

[27] J. Prins, J. Huan, B. Pugh, C. Tseng, and P. Sadayappan. UPC Implementation of an Unbalanced Tree Search Benchmark. Technical Report 03-034, UNC Dept. of Computer Science.

[28] J. Savant and S. Seidel. MuPC: A Run Time System for Unified Parallel C. Technical Report CS-TR-02-03, Department of Computer Science, Michigan Technological University, 2002.

[29] H. Subramoni, P. Lai, M. Luo, and D. K. Panda. RDMA over Ethernet - A Preliminary Study. In *Proceedings of the 2009 Workshop on High Performance Interconnects for Distributed Computing (HPIDC)*, 2009.

[30] UPC Consortium. UPC Language Specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Lab, 2005.

[31] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine. AM++: A Generalized Active Message Framework. In *Nineteenth International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sep 2010.

[32] J. Zhang, B. Behzad, and M. Snir. Optimizing the Barnes-Hut Algorithm in UPC. In *http://hdl.handle.net/2142/18699*, 2011.