The Unicorn Runtime: Efficient distributed shared memory programming for hybrid CPU-GPU clusters

Tarun Beri, Sorav Bansal, and Subodh Kumar

Abstract—Programming hybrid CPU-GPU clusters is hard. This paper addresses this difficulty and presents the design and runtime implementation of ***Unicorn*** – a parallel programming model for hybrid CPU-GPU clusters. In particular, this paper proves that efficient distributed shared memory style programing is possible and its simplicity can be retained across CPUs and GPUs in a cluster, minus the frustration of dealing with race conditions. Further, this can be done with a unified abstraction, avoiding much of the complication of dealing with hybrid architectures. This is achieved with the help of transactional semantics (on shared global address spaces), deferred bulk data synchronization, workload pipelining and various communication and computation scheduling optimizations. We describe the said abstraction, our computation and communication scheduling system and report its performance on a few benchmarks like *Matrix Multiplication*, *LU Decomposition* and *2D FFT*. We find that parallelization of coarse-grained applications like matrix multiplication or 2D FFT using our system requires only about 30 lines of C code to set up the runtime. The rest of the application code is regular single CPU/GPU implementation. This indicates the ease of extending parallel code to a distributed environment. The execution is efficient as well. When multiplying two square matrices of size 65536×65536, *Unicorn* achieves a peak performance of 7.88 TFlop/s when run over a cluster of 14 nodes with each node equipped with two Tesla M2070 GPUs and two 6-core Intel Xeon 2.67 GHz CPUs, connected over a 32Gbps Infiniband network. In this paper, we also demonstrate that the *Unicorn* programming model can be efficiently used to implement high level abstractions like MapReduce. We use such an extension to implement PageRank and report its performance. For a sample web of 500 million web pages, our implementation completes a page rank iteration in about 18 seconds (on average) on a 14-node cluster.

Index Terms—Unicorn Runtime, Distributed System Design, Scheduling, Load Balancing, Accelerators, Bulk Synchronous Parallelism

1 Introduction

Achieving high performance on modern clusters with multi-core CPUs and many-core GPUs can be tedious. Programmers have to overcome complexities arising out of heterogeneous architectures, non-uniform memory access latency, network data transfers, overlapping data communication with computations, scheduling and load balancing. Several generic frameworks like *Unicorn* [1], StarPU-MPI [2], G-Charm [3] and Legion [4] solve these problems to a varying extent. While the latter three are primarily message-passing systems, *Unicorn* provides a distributed shared memory style programming environment. This paper specifically focusses on the design, implementation and runtime optimizations required to make such a system efficient.

Conventionally, shared memory programming [5] is considered intuitive and familiar, but rather inefficient if the memory is distributed across a network. As a result, message-passing systems have gained currency in distributed programming in spite of the fact that their program structure is more complex as they leave data placement and communication mostly to the application. *Unicorn* delivers simplicity by supporting deterministic execution of distributed shared memory style programs. On the other hand, prima-facie, this simplicity imposes significant overheads on the runtime. Hence, success depends critically on a careful design of the runtime to ensure that cluster devices are kept busy by useful computation. This paper focusses on *Unicorn's* runtime and its optimizations that overcome the traditional overheads of shared memory approach. In part, this is made possible by a transactional style memory and bulk synchronicity [6]. In particular, the runtime is able to hide remote data access latency behind coarse-grained computation of work-items. It pipelines data check-out (creating private local *views* of globally shared memory), computation, and data check-in (from private *views to the shared memory*, performing lazy resolution of conflicting check-ins).

Unicorn [1] is a write-once, run-anywhere programming model. A *Unicorn* application consists of a set of inter-dependent *tasks* and can be thought of logically as BSP (Bulk Synchronous Parallel) super-steps [6]. Tasks may also hierarchically spawn other tasks. A task is ready to be scheduled at the completion of all tasks it depends on. The task graph is abstract and a program-time decision, independent of the cluster topology. Our runtime maps it dynamically to the presented cluster and executes it. A task may request any number of concurrent *subtasks*, which is a data-parallel work-sharing construct of a task, and is individually scheduled on any available device (e.g., CPU or GPU) in the cluster. Each subtask executes an application-provided "kernel function,"[1] which determines its share of

• *T. Beri, S. Bansal and S. Kumar are with the Department of Computer Science and Engineering, Indian Institute of Technology Delhi, India. E-mail: {tarun,sbansal,subodh}@cse.iitd.ac.in*

Manuscript received April 12, 2016

1. Kernels can be device-independent, or a user may provide optimized kernels for each device type

work based on its subtask ID and any task-wide parameters. *Unicorn* schedules subtasks on devices and dynamically balances their load, while also accounting for the location of their data. There is an implied barrier at the end of a task. However, the barrier's overhead is effectively mitigated by the balanced load. Further, the barrier can even be skipped by subtasks whose input dependency is satisfied by the subset of completed subtasks (of its preceding tasks).

Unicorn applications do not provide a dependency graph of subtasks (the schedulable items). Rather, they specify dependencies only among tasks. Subtasks of a task are concurrent with no inter-dependencies. This allows a natural avenue to express a higher level of parallelism. More importantly, this not only makes graphs more compact, reducing the runtime's processing overhead, but also allows more aggressive scheduling of subtasks and optimizations like out-of-order subtask execution, arbitrary subtask grouping, easy migration of subtasks across cluster nodes, etc.

Unicorn runtime supports allocation of global shared memory regions called *address spaces*. Usually a task's input and output are stored here. While an address space is logically shared, it may be physically distributed across multiple machines and devices by our runtime. An application-provided callback lists the address space regions a subtask needs to read or write. The subtask code operates only on its local copy of the data. Its memory writes are visible only to its dependent tasks, implying that any computation requiring this output must either be in the same *subtask* or in a subsequent *task*. Conflicting writes by different subtasks are resolved by an application-provided callback. We have chosen to omit any address space 'flush' or global read-/write primitive and the nature of programming simplifies significantly because of that choice. We demonstrate later that this style of programming is still efficient and powerful enough for many coarse-grained scientific applications.

Unicorn's address spaces are inspired by the idea of transactional memory. Each subtask logically checks-out its local view from global shared address space and after operating on it, the subtask checks-in its private view back to the shared address space. These private views with deferred synchronization lead to sequential consistency trivially. This also avoids several data hazards and deadlocks among subtasks, which again simplifies the application code. Thirdly, this helps our runtime perform a scheduling optimization, called *multi-assign* (section 4.3.3), where several independent instances of a straggling subtask are started in the cluster. Because of the transactional design of address spaces all private views of all but one subtasks get trivially discarded. This design also helps minimize address space coherence messages in the cluster (section 3).

Besides *multi-assign*, the runtime employs several optimizations (sections 3 and 4) for efficiency. Some of these performance optimizations target scheduling, while others focus on data transfers and minimizing control messages within the runtime. The motivation and implementation of these optimizations are the primary contributions of this paper. These optimizations enable *Unicorn* to adopt a simple distributed shared memory based programming style while retaining efficiency. This paper also explores various optimization parameters. For example, we explore various *GPU cache policies*. We study *when* to multi-assign and *how often* the later assigned device finishes first. We analyze the impact of variance in application controlled *subtask size*. We also study the impact of changing input *data availability patterns* in the cluster.

Another important contribution of this paper is to explore *Unicorn* as an extensible framework and orchestrate its functionality into other well known programming models like MapReduce [7]. We use application-provided kernel functions for the *map* stage whereas the application-provided conflict resolution is used to *reduce* two subtasks at a time. Section 5 evaluates this approach over PageRank [8] computation for a collection of web pages. The *map* stage of the experiment computes contributions of PageRank for all outlinks in the web. The *reduce* stage accumulates individual contributions on all inlinks for each webpage. Results demonstrate the efficiency of *Unicorn's* implementation and its MapReduce suitability in general.

2 Pseudo Code Sample

This section provides an overview of the *Unicorn* programming model by a pseudo code sample that multiplies two square matrices (Figure 1). More details on *Unicorn's* programming model are presented in [1]. In the pseudo code, the *matrix_multiply* function (lines 3–30) takes matrix and block dimensions as input. The first is the number of rows/columns of the square matrices. The second is the number of elements in rows/columns of each subtask. For brevity, this sample assumes that *matrix_dim* is divisible by *block_dim*. The sample first registers *data subscription* and OpenCL based *subtask execution* callbacks (lines 6–7), and then creates input and output address spaces (lines 13–15), which are bound (lines 24–25) to the matrix multiplication task submitted to *Unicorn* runtime for asynchronous execution (line 27). The purpose of *data subscription* callback is to indicate access patterns and input data distribution among subtasks. On the other hand, *subtask execution* callback specifies the subtask execution logic. Each of the subtasks executes *matmul_subsciption* function to subscribe to their input (or output) data (lines 50–52). The actual subscription information can be specified as (offset, length) pairs for contiguous subscriptions or in a quad-tuple format – (offset, length, step, size) for block based subscriptions (lines 46–48).

In the example, we have omitted the core subtask logic, for it is nothing but publicly available OpenCL implementation of matrix multiplication. Note that *Unicorn* subtasks can be written more efficiently in languages closer to hardware (i.e., CUDA or C/C++). However, for ease of programming, we support the high level abstraction of OpenCL which allows execution of same subtask code on CPUs and GPUs.

Unicorn is designed to treat data and tasks independently. Thus, the lifetime of address spaces is explicitly decoupled from tasks. This allows address spaces to outlive and be used across multiple tasks. An address space can be bound to a task in three modes – read-only, write-only and read-write. The first one is useful for data sharing across multiple concurrently running tasks. The other two, however, need creation of private local views of subtasks and synchronization for resolving conflicting writes. Note that *Unicorn* allows concurrent tasks if they do not share address spaces or the shared address spaces are bound in read-only mode.

```
1  struct matmul_conf { size_t matrix_dim, block_dim; };
2
3  matrix_multiply(matrix_dim, block_dim)
4  {
5   key = "MATMUL";
6   register_callback(key,SUBSCRIPTION, matmul_subscription);
7   register_callback(key, OPENCL, "matmul_ocl", "prog.ocl");
8
9   if(get_host() == 0)  // Submit task from single host
10  {
11   // create address spaces
12   size = matrix_dim * matrix_dim * sizeof(float);
13   input1 = malloc_shared(size);
14   input2 = malloc_shared(size);
15   output = malloc_shared(size);
16
17   initialize_input(input1, input2); // application code
18
19   // create task -- one subtask per output matrix block
20   block_count = matrix_dim / block_dim;
21   subtasks = block_count * block_count;
22   task = create_task(key, subtasks, matmul_conf(matrix_dim
          , block_dim));
23
24   bind_address_spaces(task, READ_ONLY, input1, input2);
25   bind_address_spaces(task, WRITE_ONLY, output);
26
27   submit_task(task);
28   wait_for_task_completion(task);
29  }
30 }
```

```
31 matmul_subscription(task, device, subtask)
32 {
33  matmul_conf* conf = (matmul_conf*)(task.conf);
34
35  block_count = conf→matrix_dim / conf→block_dim;
36  block_row = (subtask.id / block_count);
37  block_column = (subtask.id % block_count);
38  matrix_row_size = conf→matrix_dim * sizeof(float);
39  block_row_size = conf→block_dim * sizeof(float);
40  block_row_offset = block_row * conf→block_dim *
        matrix_row_size;
41  block_column_offset = block_column * conf→block_dim *
        sizeof(float);
42  block_offset = block_row_offset + block_column_offset;
43
44  // Specify subscriptions - (offset, size, step, count).
        First input matrix subscribes to all blocks in the
        row block_row, second input matrix subscribes to all
         blocks in the column block_column, output matrix
        subscribes to the block at (block_row, block_column)
45
46  block_subscription_info bsinfo0(block_row_offset,
        matrix_row_size, matrix_row_size, conf→block_dim);
47  block_subscription_info bsinfo1(block_column_offset,
        block_row_size, matrix_row_size, conf→matrix_dim);
48  block_subscription_info bsinfo2(block_offset,
        block_row_size, matrix_row_size, conf→block_dim);
49
50  subscribe(task.id, device.id, subtask.id, 0, bsinfo0);
51  subscribe(task.id, device.id, subtask.id, 1, bsinfo1);
52  subscribe(task.id, device.id, subtask.id, 2, bsinfo2);
53
54  // OpenCL kernel launch configuration
55  set_launch_conf(task.id, device.id, subtask.id, ...);
56 }
```

Fig. 1: Unicorn program for Matrix Multiplication

3 Shared Address Spaces

Unicorn applications can request the runtime to allocate address spaces. A task binds itself to the address spaces it plans to access, specifying *read-only*, *write-only* or *read-write* intent. All subtasks of the task may then access the bound address spaces. Address spaces may be shared by multiple tasks. Two examples are: a task's output (write-only) address space bound as input (read-only) by a subsequent task, or an address space containing constant data serving as input to a series of tasks. Address spaces are generally distributed with their data spread across multiple nodes.

Subtasks (of tasks) operate upon their bound address spaces by means of subscriptions. *Read* subscriptions define address space bytes required for computations done by the subtask and *write* subscriptions define address space bytes produced by the subtask (as output). These bytes can form a single contiguous region of memory within the address space or a set of uniformly or randomly distributed contiguous memory regions. A subtask may register any number of such subscriptions with the runtime. Each subscription is specified using the quad-tuple (offset, length, step, size). This format enables applications to specify generally used contiguous, block based and strided subscriptions in very few calls. It also reduces the number of subscription calls the runtime processes per subtask. Block and strided subscription are similar to MPI_Type_vector [9].

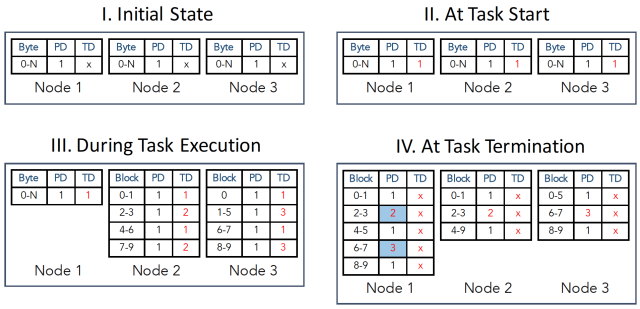


Fig. 2: Address Space Ownerships – PD represents Ownership Directory, TD represents Temporary Ownership Directory, x represents non-existent directory, red text represents changes from last state and light blue cells represent directory changes via explicit ownership update messages

As tasks execute and their subtask subscriptions are processed, address spaces are logically fragmented into memory regions defined by these subscriptions. Internally, address spaces store these logical fragmentations (called *data regions*) in terms of their quad-tuples. Each of these *data regions* have an associated *owner node* in the cluster and this *owner node* notionally contains the entire data corresponding to the *data region*. The mapping of *data regions* to corresponding *owner nodes* is stored in a directory called the *address space ownership directory*, or simply the directory.

An application may create any number of address spaces from any node in the cluster. The node on which an address space is incarnated is called its *creator node*. Internally, *Unicorn* also assigns a *master node* to every address space created by the application. As explained below, the *master node* serves as an intermediary to enable efficient management of the directory. *Master nodes* are chosen in round robin fashion to balance the load of all address space routing requests among all cluster nodes. The master node contains the master copy of the directory for a given address space. Other nodes using that address space generally contain a partial map: a subset of *master ownership directory*. If these other nodes require a region that exists in their partial map, they directly send region fetch requests to the corresponding owner. On the other hand, if a region is not present in the partial map, they route address space fetches through the master node. The master then consults its ownership directory and forwards the request to the actual owner containing the data region in question.

When an address space is created, it contains a single region. The ownership directory on all nodes is updated to map this region to the selected master (state I of figure 2). As a subtask executes and commits its writes (originally in its private view) to the address space, the node on which the subtask executed becomes the new owner of the data regions write-subscribed by the subtask. (For multiple writers,

the final owner is where the final reduction occurs). At the end of a task, the new owner node records this ownership in its directory and also sends this ownership update to the master (state IV of figure 2). If the master observes a new owner node, it updates its own map and forwards the ownership update message to the previous owner(s), which also updates its map. All other nodes always initialize their maps to point to the master at the end of the task. Thus, at the beginning of a task all nodes map to themselves data regions they own and point to the corresponding master for all other data regions. The master, however, always knows the true locations of all data regions in the address space.

Unicorn tasks guarantee transactional semantics. This means that address space ownership updates are reflected only at task boundaries and all subtasks of a task see the same address space data during task execution. In other words, even for an address space marked *read-write*, where a few subtasks are updating the address space while a few are reading it, the ones that are reading must not see the new data but read the state at the inception of the task. Our address spaces achieve transactional semantics simply by deferring address space ownership updates to the end of the task. Since ownerships are not modified during task execution, all subtasks continue to refer to the original data location until task boundary. The following sequence of steps define our delayed ownership update semantics –

1) All nodes hold information about write subscriptions of all locally committed subtasks in a local data structure. This information is a set *'S'* of quad tuples (offset, length, step, size).
2) At the end of the task, all nodes commit *'S'* into their *address space ownership directories* and also send an ownership update message to the *master node*. This message also carries the set *'S'*.
3) The master commits the received set *'S'* from every other node. It records the *data regions* that have changed owners (during the task) and sends the previous owners another ownership update message.
4) All non-master nodes process the ownership update message from the master and record master as the new owner of the *data regions* in the message.

Note that we maintain one set *'S'* per master node. For example, if a task employs three address spaces with two having the same master and the third having a different master, we maintain only two sets, one for each master.

Often, read subscriptions of subtasks (of a task) overlap. In such cases, it is prudent that data once fetched (by a subtask) not be re-fetched when requested by another subtask on the same node. To accomplish this, our runtime records all data fetched on a node in a separate address space directory. This directory is temporary as its lifetime is bound to that of the ongoing task and is thus called *temporary ownership directory*. At inception, the temporary directory is a logical replica of the *address space ownership directory* on every node (state II of figure 2). Specifically, as data is fetched, it records the updates (state III of figure 2). For every read subscription request from a subtask, the temporary directory is the one consulted and not the main directory. This ensures that no data is transferred again during the lifetime of the task. Note that this mechanism also conforms to the transactional semantics guaranteed by our runtime. It further allows several nodes to be simultaneously designated temporary owners of a data region without the need of an explicit handshake (or ownership update message) for data sharing.

At the end of the task, the data fetched for reading (and recorded in temporary directory) may become stale as data ownerships could have changed and the new data owner could have written new data in that region. Thus, at task boundaries, the temporary directories are simply discarded. However, if a task does not write to an address space (and subscribes to it read-only), as an optimization its temporary directory entries are retained for the subsequent task.

To analyze the overhead of our ownership update protocol, we consider a task executing on an N node cluster and using K address spaces, $K > N$. Our master selection mechanism ensures that every node is master of at least one address space. Let's assume that subtasks on node j make W_{ij} write subscriptions for address space i. Thus the total number of write subscriptions in the cluster is equal to

$$M = \sum_{i=1}^{K} \sum_{j=1}^{N} W_{ij}$$

Updating ownerships at the first instance would mean sending (or perhaps broadcasting) at least M ownership update messages in the cluster. However, by delaying these messages to task boundaries and by sending only one ownership update message per *master node*, we reduce the number of ownership update messages generated in the cluster. In this example, every node being master of at least one address space receives *N-1* ownership update messages and every master sends out a similar number of messages (in the worst case) to non-masters. Thus, the maximum number of ownership update messages possible with our protocol is *2N(N-1)*, which is significantly fewer than *M*.

Besides transactional semantics, the protocol ensures efficiency as no explicit ownership update messages are required during task execution. Only at the end of the task, a few messages are exchanged on the change of ownership of any data region. To further limit the number of such messages, we aggressively reduce address space fragmentation by combining adjacent directory records. Specifying subscriptions in terms of block regions further reduces the number of directory entries. Note that we do not employ the usual MSI coherence protocol (where cache lines are explicitly tagged to be *Modified*, *Shared* or *Invalid*) as it has the potential to generate too many cache invalidations.

4 Runtime System

This section describes the design of *Unicorn's* runtime along with the motivation for design choices and trade-offs. We also discuss optimizations that enable our runtime.

4.1 Runtime Components

On initialization, *Unicorn* starts one MPI process per node in the cluster. This process begins by creating an instance of the *controller* (Figure 3). The controller manages the creation, destruction and lifetime of all other runtime components. These include the *Profiling Manager* and two subsystems of the runtime – *network subsystem* and *scheduling subsystem*. The controller is also the interface between the application

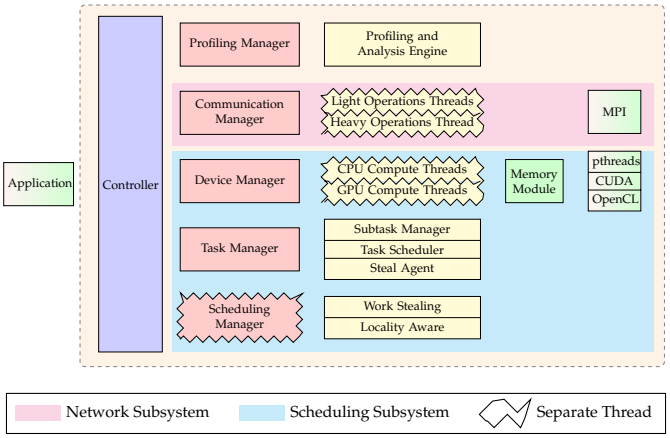


Fig. 3: Unicorn runtime – Light orange region represents the instance of the runtime on each node in the cluster

and rest of the runtime. All application requests like creation and submission of tasks, creation and destruction of address spaces, etc. pass through the controller. User space callbacks are also executed through the controller.

Once the controller initializes other runtime components, they directly talk to each other without the controller's involvement. All modules store diagnostic information with the *Profiling Manager*. The information collected by the *Profiling Manager* on all nodes is accumulated on MPI master node at application shutdown. This data may be dumped to stdout/stderr or may be sent to *Profiling and Analysis Engine* for further processing. The engine also converts the data into readily consumable graphical and tabular formats for performance debugging and diagnosis. More details on this are beyond the scope of this paper.

The *network* and *scheduling* subsystems comprise threads that manage cluster-wide operations like subtask scheduling and data exchange between nodes. All control messages (generated by the runtime) and all data transfers (requested by executing subtasks) destined for remote nodes are routed through the *Communication Manager*, which filters duplicate requests made by various subtasks and also combines multiple requests targeted for same destination node, whenever possible. The *network subsystem* is a collection of three threads and uses MPI_THREAD_MULTIPLE. One of these threads is dedicated to subtask data transfers across nodes which are classified as *heavy operations*. The other two are designed for relatively *light operations* and respectively handle *fixed size requests* and *variable size requests*. Fixed size requests are the ones whose data size is pre-known and special MPI optimizations that re-use the same data buffers repeatedly are employed. Variable size data requests have varying lengths and buffers are not pre-allocated for those. Rather they are handled using *MPI_Probe* in a separate thread. Segregating heavy and light operations also allows critical control messages in the runtime to be transferred quickly. This indirectly helps asynchrony by keeping most runtime threads active rather than waiting on a few commands.

The *scheduling subsystem* is a two-level hierarchy of threads with "scheduler thread" employing a "compute thread" for every device (i.e., CPU core and GPU) on the node. The *Device Manager* manages these *compute threads*, each of which have an exclusive priority queue. All commands that are targeted for execution on a device are enqueued in the corresponding priority queue. Each *compute thread* has a helper *Memory Module*. For CPU compute threads the module manages sharing of read-only address space data and creation/destruction of virtual memory for local working copies of the address space subscriptions of the subtasks. For GPU *compute threads*, the module employs a software cache for efficient sharing of scarce GPU memory between subtasks. It also manages the creation and destruction of pinned memory buffers used for bidirectional Direct Memory Access (DMA) transfers between CPU and GPU.

The *Task Manager* enqueues each task submitted by the application and submits them for execution (to the *Scheduling Manager*) when all its dependencies are fulfilled. This is accompanied by creation of a *Subtask Manager*, *Task Scheduler* and *Steal Agent*. The *Subtask Manager* keeps track of the subscriptions and data transfers of each subtask executing on the node. It also tracks the subtask execution times, which helps estimate the relative execution rates at different cluster devices. This also helps in determining if a subtask is straggling and if it should be multi-assigned to some other device. The *Task Scheduler* co-operates with the *Subtask Manager* and the *Scheduling Manager* for executing a subtask and collecting its acknowledgement. The *Steal Agent's* role is limited to directing an incoming steal request to the device with the highest load, thereby reducing the number of steal attempts in the cluster (section 4.3.2).

The *Scheduling Manager* handles scheduling of all tasks submitted by the application. Like *compute threads*, it has a dedicated thread backed by an exclusive priority queue. The *Task Scheduler* of every task enqueues subtasks in the *Scheduling Manager's* queue and it schedules them for local or remote devices (using the *Communication Manager*). When a device finishes execution of its subtasks, an acknowledgement is sent to the corresponding *Task Scheduler* through the *Scheduling Managers* on various nodes communicating via their respective *Communication Managers*.

The default *Unicorn* scheduler (section 4.3) is locality-oblivious and based upon two-level work stealing assisted by *Steal Agents*. However, a task may opt for locality aware scheduling (section 4.3.4), in which case data locality (on cluster nodes) is incorporated into scheduling decisions. Having a different system-wide *Scheduling Manager* and a *Task scheduler* per task allows *Unicorn* to employ different scheduling policies for different tasks running at the same time. However, the common functionality is abstracted out into the system-wide *Scheduling Manager*.

4.2 Network Subsystem

Distributed systems are often bottlenecked by communication. Efficiently transferring data across the network is critical for sustained performance in a cluster computing environment. Our network subsystem is optimized for performance and works closely with *compute threads* and *address spaces*, merging duplicate and overlapping requests, reducing both request and returned data message counts and sizes. Besides, subtask data fetch requests are given high priority while the prefetch requests for subtasks anticipated to run in the future are given lower priority. The subsystem supports lossless compression of transferred data (section 4.6). This is particularly important for large data transfers.

Recall that concurrently executing *Unicorn* subtasks may share part of their input data by subscribing to overlapping

regions of input address spaces. For each subtask subscription (specified as (offset, length) pairs or quad tuples (offset, length, step, count)), our network subsystem queries the address space directory for the corresponding data location in the cluster. The query response may point to a local data location if the data is resident locally on the node. In case the data is not locally resident, it may already be enroute due to an earlier request or a new request must be initiated. In either case the memory module returns a unique handle (internally implemented like pthread signal-wait) on which the calling *compute thread* can wait till the data arrives.

The network subsystem initiates a remote data fetch against this handle if the request does not fully overlap with previous requests. An R-tree [10] is used for overlap detection. When the data fetch completes, the handle is notified and the waiting thread is signaled. Address space query and handle creation happen atomically, ensuring that any other *compute thread* requiring the same data is returned the same handle to wait on. In a typical scenario, a *compute thread* may wait on several such handles and a handle may be waited on by several *compute threads*.

In our implementation, a subtask first issues all subscription requests and collects all the remote handles it depends on and then an aggregate handle is created which it actually waits on. The aggregate handle is unique to a *compute thread* while the internal handles may be shared by several *compute threads*. Handles created for remote data fetch for a subtask may be destined for one or more nodes. Our network subsystem combines them into a single request to the data owner. The response, however, is separate for different handles, as this helps earlier awakening of *compute threads* waiting upon a subset of handles.

In addition, message count is also reduced by piggybacking non-urgent control messages on other messages. For example, address space ownership update messages (section 3) are combined with subtask completion acknowledgements. This reduces network congestion making way for higher priority messages.

Often subtasks access data in patterns. This is especially true of regular coarse-grained experiments where a subtask may access multiple contiguous ranges of data with uniform separation. In such cases, our runtime detects access patterns (like block or strided) using the R-tree and instead of issuing multiple remote data transfer requests for each subtask, one unified request is issued. At the remote end, the data is packed before being sent to the requestor where it is unpacked and mapped into the requesting subtask's local view. This optimization helps scalability (by preventing flooding of the network with too many small data transfer requests) and simplifies application programs (which do not need to consider data and communication granularity).

4.3 Scheduling Subsystem

Unicorn's scheduling subsystem is both proactive and reactive. It factors current load when scheduling each subtask. With substantial variance in system loads and coarse grained subtasks' loads however, even one job can impede the entire application. Hence our scheduler adapts, reassigning lagging jobs.

Unicorn employs a work stealing scheduler [11], [12] to balance load on various cluster devices. In work stealing, a device (called *stealer*) that has exhausted all its work selects another device (called *victim*) with potentially outstanding work, requesting a share of its work so that both can together achieve faster completion. Although work stealing (with random victim selection) has proven to be quite effective in several parallel systems, it poses several challenges in our context of hybrid CPU-GPU clusters. First, care is needed to limit the overhead and latency of stealing. Secondly, the effectiveness of work-stealing tends to reduce as computational disparity and heterogeneity (memory hierarchy, device architecture (x86_64/Fermi/Kepler), programming styles (OpenCL/CUDA/C++)) between devices grow. CPUs allow threads to be scheduled on a single core, but GPUs do not allow per-core scheduling and kernels can occupy the entire GPU[2]. Subtasks large enough to effectively use the GPU can be too slow on the CPU. Shorter ones may improve CPU performance, but GPUs remain underutilized. In our tests, a BLAS [13] based sequential implementation of multiplication of two dense square matrices with 8K elements each (on Intel Xeon X5650) was outperformed by a CUBLAS [14] based implementation (on Tesla M2070) by a factor of 33x+. Similarly, a BLAS based block LU-Decomposition on the GPU reported 10x+ speedup over the sequential implementation. This wide disparity between CPUs and GPUs poses scheduling challenges.

Our scheduler makes subtask assignments in groups. Subtasks in the group are sequentially executed starting with the first one. While a device is executing the first subtask in the assigned group, the next subtask overlaps its communication with the ongoing computation of the first subtask. Similarly, the third subtask overlaps its communication with the second subtask's computation. This continues for all subsequent subtasks in the group. This mechanism is especially useful for GPUs capable of compute-communication overlap and multiple kernel launches, where we create a pipeline of subtasks. At any given time, one subtask may be transferring its data to the GPU, one or more subtasks may be executing and one subtask may be copying its data out of the GPU.

4.3.1 Handling CPU-GPU Performance Disparity

The CPU-GPU performance disparity can lead an application programmer to design work-units differently for different devices. But this would compromise abstraction and simplicity of programming. In our framework, applications logically partition tasks and remain unaware of where each subtask is scheduled. Dynamically adapting subtask sizes to each execution environment can be tedious for an application. An option is to group, say, multiple CPU cores together into a single device, but this can become complicated with a large number of device types with diverse capabilities. Additionally, this would also compromise the simplicity of the programming model as each kernel must execute on a specific "set of devices." This also eliminates the liberty to use existing sequential CPU functions as subtask kernels, an important design goal for us. Another alternative is to envision a subtask as a set of work-items and schedule a single work-item per CPU core. This is analogous to OpenCL's [15] work-group and work-item.

2. We do not consider kernel fusion, allowing the user to devise the most efficient kernel.

Each work-item is intended to run on one CPU core of the work-group and the maximum size of the work-group created for a subtask is equal to the number of CPU cores on a node. The final size of a work-group (for subtasks of a task) is, however, computed using binary search over the range [1, number of CPU cores]. Our algorithm requires at least two subtasks to bootstrap the binary search. For the first subtask, the size of work-group is set to the number of CPU cores on the node and for the second subtask the size of work-group is set to half the number of CPU cores. Next, we compare the execution times of both these subtasks. If the former one's execution time is shorter, the size of work-group for the third subtask is set to three-fourth the number of CPU cores. On the other hand, if the latter execution time is longer, then the size of work-group created for the third subtask is one-fourth the total number of CPU cores. This dynamic adjustment continues until the end of the task.

This dynamic calibration ensures that changing system load is effectively dealt with and CPU cores do not over-execute subtasks. Over-executing can be detrimental not only to the subtasks executed by CPUs but to the ones executed by GPUs as well. First, because several CPU cycles are required to complete GPU's work. If CPUs remain busy with other work, they won't be able to sufficiently keep the GPU pipeline busy by moving data to/from pinned memory (and GPUs) and by launching GPU kernels. Further, large work on these slower cores also unnecessarily overload the memory and network throughputs at these nodes.

By resizing subtasks at runtime, *Unicorn* is able to support a wide disparity among devices' computation powers. For example, on a node with, say, 16 cores, a CPU subtask may be decomposed into up to 16 smaller units, each scheduled on one core. On the other hand, GPU subtasks may be logically grouped into bigger units. Multiple GPU subtasks run concurrently using CUDA streams [16]. With, say, 16 streams a disparity of 1:256 in the load of subtasks can be supported. Results in section 5.5.2 show our runtime's effectiveness in handling such skew on a variety of subtasks.

4.3.2 Work Stealing in Unicorn

The benefit of dynamic scheduling is that it responds not only to varying subtask load and compute unit capacity but also to varying network throughput. *Unicorn* scheduler begins a task by assigning an equal number of subtasks to each available device. Once a device nears completion, it steals subtasks from another device. We employ two-level random stealing, where a stealer first tries to steal from a busy device on the same node. On failure, it requests the steal-agent on a randomly selected remote node, which in turn attempts to steal from its local devices. Although one might extend shared-memory algorithms to stealing over shared address spaces or one sided MPI communication [12], we find the overheads too high for that. On the other hand our runtime is already based on multiple asynchronous threads on each node and it is easier to use that runtime infrastructure for load stealing without incurring loss in compute threads. Our pipelined schedule implies that the subtasks may be

1) idle,
2) waiting for data,
3) ready for execution, or
4) executing.

Our lock-free steal algorithm allows steal from any stage from idle to ready for execution. (Steals from stage 4 is allowed as a special case discussed in section 4.3.3). The steal-agent on each node helps accomplish this as it maintains the count of outstanding subtasks in each stage for all local devices on its node and also monitors their execution rates. (Since we do not model subtask heterogeneity, we simplify by using the past rate as a predictor for future rate).

We limit the stealer's capability to steal from any stage of the pipeline by associating an aggression level (a measure of the stealer's execution speed and idle time while waiting for steal) with every device. We keep GPUs at the maximum aggression level (3) while the aggression of CPUs increases with the number of consecutive failed steal attempts. CPUs start with aggression 1 and gradually increase to 3 after $N/2$ consecutive steal failures, where N is the number of nodes executing the task. Note that each device records the number of steal requests made by them and their outcomes (success or failure). Thus, our two-level victim selection technique proceeds as follows for device d_{ij} on node i:

1) In first attempt set victim $v = i$, otherwise, choose random victim v, different from victims selected in this round. (Round resets when a victim is found or no potential agents remain.)
2) Send steal request to the steal agent of node v. The steal request includes the measured subtask execution rate, R_{ij} of d_{ij} and its aggression level a_{ij}, a value in range [1,3].
3) Agent v selects device d_{vw} that is expected to complete its queue the last among all devices of node v. (i.e., the highest value of $\frac{Q_{vw}}{R_{vw}}$, where Q stands for the length of the queue at the given device). A subtask in stage k is counted in Q only if $a_{ij} >= k$.

To further reduce the number of steal attempts in the cluster, a range of subtasks is stolen. We compare the devices' execution rates to decide the quantum of the final steal: $\lceil Q_{vw} \times \frac{R_{ij}}{R_{ij}+R_{vw}} \rceil$. If this is less than 1, steal fails.

In contrast, a one level stealing algorithm, where a device directly steals from another randomly selected device, generates too many steal requests and is not scalable. For example, if we assume N nodes and D devices per node, and that at a time t, every device has a probability p_t of having an empty queue. The probability of a one level random steal attempt being successful is $\frac{1-p_t}{N \times D}$, while that of steal from a random steal agent is $\frac{1-p_t^D}{N}$. Our experiments described in chapter 5 also demonstrate this improvement. We do not see the benefit of a deeper hierarchy (e.g. multiple steal-groups on a node [17]) due to its management overheads. However, for larger clusters it is possible that organizing nodes into hierarchical steal-groups will be useful.

Work stealing in *Unicorn* is light weight, as only the subtask ID is 'stolen' and not the associated data. Further, *Unicorn* uses two techniques to limit the overhead of stealing. First, it employs a steal-agent per node in the cluster. Due to shared memory usage, the agent is able to easily monitor local load and track the ability of each local device to service an incoming steal request. This guided victim selection helps making more targeted steal requests, thereby reducing the number of unsuccessful steal attempts in the

cluster (section 5.3). The selection of a victim node for a steal request remains random.

Secondly, *Unicorn* employs a proactive stealing technique called *ProSteal* [18]. When a device runs out of work (i.e., subtasks), its task pipeline stalls. It can no longer overlap communication of subtasks with computation of others. It needs to prime its task pipeline afresh after it steals a new subtask. *Unicorn* prevents this performance loss in the system by stealing early (especially for GPU devices) with one or more subtasks still pending with the device. The exact number of subtasks pending with a device at the time of steal is computed dynamically based on the device's rate of subtask execution and the observed latency to prime the pipeline after a stall.

4.3.3 Multi-Assign: Handling Stragglers

Unicorn scheduling is non-preemptive, i.e., there is no migration of a running subtask, along with its running state, from one device to another. If a subtask takes much longer than others to finish, *Unicorn* may *multi-assign* that subtask, i.e., assign the same subtask to another device [19], [20]. At this stage, multiple instances of the same subtask may be running in the cluster; there is little benefit of killing it at the straggler. By this stage, all remaining subtasks have been stolen away from that device. Hence, the instance that finishes first commits its results to the global address space. Other running instances of the subtask are cancelled and their output (in their private views) is discarded.

We employ a simple protocol to determine the subtask instances to be cancelled. The list of all subsequent assignees is maintained at the original assignee and each subsequent assignee records the ID of the original. When a device finishes, it informs the original assignee, which in turn sends cancellation messages to all others. The commit is performed by the finisher if it receives the cancellation message with its ID.

Multi-assign is implemented as a part of stealing. In particular, we allow a subtask in 'executing' stage to also be stolen, while it is allowed to continue executing. Such steals are only allowed when the steal request specifically includes a multi-assign flag (i.e., the stealing device is at maximum aggression level (section 4.3.2)) and the victim is not a multi-assignee. In this case, the victim is marked as the original assignee and the stealer as a multi-assignee.

We multi-assign only if the stealing device's estimated completion time is less than the victim's *remaining* estimated completion time. The estimated completion time of a device is based on its observed execution rate and observed average data fetch time per subtask. We use the remaining time for victim because it has already executed a part of the subtask at the time of *multi-assign*. Thus, the time it has already spent in execution of the subtask is subtracted from its total estimated time. As an additional heuristic, a subtask is only multi-assigned to a device on a different node or to a different device type on the same node (CPU subtasks are multi-assigned to GPU and vice versa). This reduces the chance of similarly slow execution on the re-assigned device. Experiments show significantly faster completion times due to multi-assignment. This is especially useful in networked environments, where the effective data transfer bandwidths may be variable and a device on a node with slow link may not receive its input fast enough, delaying the entire task (and its dependents).

4.3.4 Locality Aware Scheduling

Often a coarse-grained computation is decomposed such that adjacent subtasks exhibit spatial locality and access adjacent regions of input address spaces. However, this is sometimes infeasible or it is overly complicated to write programs in this fashion. To help such application programs, our runtime maintains a distributed map of data resident on various nodes (section 3) and uses it to estimate the affinity of work to different nodes to guide scheduling. Traditionally, locality-aware scheduling has mostly focused on maximizing reuse of resident data. In contrast, *Unicorn* also focuses on scheduling subtasks such that the cost of fetching the non-resident data is minimized. This is because significant time can be lost in fetching remote data and different devices are able to consume data at different rates. More information on this can be found in [21].

4.4 Software GPU Cache

Moving data between CPUs and GPUs is a relatively expensive operation. For performance reasons, it is important to reduce the frequency and volume of such data copies. Subtasks of a *Unicorn* task are allowed to have overlapping subscriptions. In such situation, it is beneficial to transfer data from CPU to GPU only once and use it for as many subtasks as possible. Similarly, output produced by a *Unicorn* task may serve as input to a subsequent task. In this case, data can be retained on GPUs at the end of the task in anticipation that a subtask (of a future task) may be scheduled on the same GPU.

Our runtime maintains an on-GPU software cache to optimize DMA transfers to the device. The cache allows a GPU to skip a data transfer in case the data is already resident. This scheme is suitable for both the scenarios described above, i.e., multiple subtasks of a task share read-only data and data written by a subtask of a task is later read by a subtask of another task. An exclusive instance of GPU cache is created for every GPU in the cluster.

Unicorn's runtime supports five GPU cache eviction policies – LRU (least recently used), MRU (most recently used), LFU (least frequently used), MFU (most frequently used) and Random. The eviction policies come into play when GPU has low memory, which needs to be freed up by evicting one or more cache entries. By default, our current implementation employs LRU cache eviction policy. Depending on the nature of the application, other cache eviction policies may be requested. A study of performance implications of these policies is presented in section 5.

Unlike hardware caches, we do not have a concept of cache lines that get invalidated. Rather, we invalidate entire subtask subscription. A study on a true cache implementation versus ours may provide pointers for a better cache design. This is beyond the scope of this paper.

4.5 Conflict Resolution

Unicorn subtasks may subscribe to overlapping regions in a writable address space. This is treated as a write conflict among subtasks and the application must explicitly specify a reduction mechanism to resolve the conflict. Otherwise, the output is undefined. *Unicorn* runtime provides a set of

built-in reduction operators. Applications are also free to define custom reductions.

Unicorn treats reduction as a first-class citizen and the outputs of subtasks are aggressively reduced as soon as they are ready. Thus, at any given time, a few subtasks may be in the execution stage while a few others may be in reduction stage. Aggressively performing reductions helps in reducing overall execution time as these are executed during the time subtasks await their remote subscriptions to be fetched. Secondly, this reduces memory pressure on the system. As soon as two subtasks reduce, the memory of one of these is freed. Steal attempts are made only if there is no reduction to perform. GPU kernels are by design multi-threaded. To efficiently perform reduction, *Unicorn* employs OpenMP's *parallel for* construct to speedup execution of built-in reduction operators. Further, before transmitting data to remote nodes for reduction, it is compressed by our network subsystem. We also compress data before transmitting from GPUs to CPUs (section 4.6). *Unicorn* provides a few built-in compression algorithms but applications may also provide their own.

4.6 MapReduce Extension and Data Compression

The MapReduce [7] programming model visualizes a task in two stages. The first stage (called *map stage*) marshals the input data into different groups while the second stage (called *reduce stage*) consumes these groups and produces a reduction (or summarization) of each. Both stages are individually parallelizable and can be run distributedly over a cluster. The reduce stage, however, must start after the map stage is complete.

In this paper, we explore *Unicorn's* suitability to realize this high level programming abstraction. For the map stage, we deploy *Unicorn's subtask execution* callback while the reduce stage is executed by *data reduction* callback. Note that these two callbacks are already executed sequentially by *Unicorn's* runtime. In the former callback, subtasks concurrently process disjoint data (subscribed by of *data subscription* callback) from input address space(s) and produce a logical grouping in output address space(s). For example, in PageRank experiment (section 5), the callback results in each subtask producing an array whose indices represent web page IDs and values are real numbers that represent page rank contributions from the data (or web pages) processed by this subtask. For the reduce stage, our runtime takes two subtasks at a time and logically sums them up (i.e., adds the page rank values at every index of the output of both subtasks). Note that Unicorn provides built-in functions for mostly used reductions operations like summation.

In such experiments, however, the data produced by map stage is sparse because a subtask contributes page ranks to only a small fraction of the web. Thus, most indices in a subtask's output array (in the address space) are zero. In a cluster environment like *Unicorn*, data reduction requires movement of a lot of data. This includes both inter-node data transfers and GPU to CPU transfers. Our runtime provides a few simple compression routines like *Run Length Encoding* (RLE) where a sequence of frequently occurring values (like zero) in a sparse array are replaced by their run lengths. Results in section 5 provide more insight into the performance benefits of this scheme.

5 Experiments

We have implemented several coarse-grained scientific computation benchmarks over *Unicorn*. These include image convolution, matrix multiplication, LU matrix decomposition, two-dimensional fast Fourier transform (2D-FFT) and PageRank. These benchmarks have well known parallelizations and are diverse enough for studying different kinds of complexities. Image convolution is embarrassingly parallel, but each subtask needs a fringe around its tile, which is produced by another subtask. Matrix multiplication is computationally intensive with heavy data transfer requirements. LU decomposition has a nested task hierarchy, 2D-FFT involves a parallelization-unfriendly matrix transpose operation and finally PageRank is a MapReduce based computation. The goal of these experiments is to assess conditions under which our runtime responds well.

Our experimental cluster has fourteen nodes, each equipped with two 6-core Intel Xeon X5650 2.67 GHz processors and two Tesla M2070 GPUs. The nodes run CentOS 6.2 with CUDA 5.5. For communication, we use Open MPI [22] 1.4.5 over an InfiniBand [23] network with 32Gbps peak bandwidth.

Our runtime allows restricting applications to only certain cores, e.g., only CPU cores, only GPUs, or both, However, CPU cores, being main OS vehicles, are not used purely for computation but also other support functions like CPU-GPU data transfers, GPU kernel launches, network data transfers, etc. When subtask data usage is high (e.g., in several hundred MBs per subtask), significant CPU load is incurred for CPU↔GPU data transfers. Based on our empirical analysis, we withhold up to two CPU cores (per node) from subtask execution when both CPUs and GPUs are requested by an application. In section 5.7.1, we study the performance impact of varying the number of CPU cores employed in computations. This also provides an insight into the library's runtime overheads.

Many of our experiments use subtasks of size 2048×2048. This empirically determined size works well for both CPUs and GPUs. The size is not too large for CPUs (to cause frequent cache misses) and not too small for GPUs (to cause their under-utilization). In section 5.5.2, we present the impact of varying subtask size on performance.

We first discuss the implementation of the selected benchmarks over *Unicorn* and then present how these scale with an increasing number of nodes. We also evaluate *Unicorn's* work-stealing, runtime optimizations like *multi-assign*, *pipelining* and *GPU caches*, and overheads. Unless stated otherwise, the input address spaces are randomly distributed (as 2048×2048 blocks) over the cluster nodes and our runtime transparently moves data on-demand to other nodes, as the program executes. Note that all presented measurements are minimum of at least three trials.

5.1 Unicorn Parallelization of Benchmarks

In our image convolution experiment all color channels of a $2^{16} \times 2^{16}$ 24-bit RGB image are convolved with a 31×31 filter. The input image is stored in a read-only address space (initially distributed randomly across the cluster), logically divided into 1024 blocks of size 2048×2048. Each subtask convolves one block. Because convolution at boundaries

requires data from adjoining blocks, the input memory subscription of a subtask overlaps with other subtasks', usually at all four boundaries. The output image is generated in a write-only address space.

In the matrix multiplication experiment two dense square matrices of size $2^{16} \times 2^{16}$ each are multiplied to produce the result matrix. Each input matrix is stored in a read-only address space and the result matrix is stored in a write-only address space of the task. The output matrix is logically divided into 2048×2048 blocks and computation of each block is assigned to a different subtask (which subscribes to all blocks in the corresponding row of the first input matrix and all blocks in the corresponding column of the second input matrix). The CPU and GPU subtask callbacks are implemented using single-precision BLAS [13] and CUBLAS [14] functions respectively.

For the in-place block LU Decomposition [24] experiment, the input matrix ($2^{16} \times 2^{16}$) is kept in a read-write address space and is logically divided again into 2048×2048 sized blocks. The matrix is solved top-down for each diagonal block. For a matrix divided into $n * n$ blocks, solving for each diagonal block (i, j) involves three tasks – LU decomposition of the diagonal block (i, j), propagation of its results to other blocks in its row $(i, j+1...n)$ and column $(i+1...n, j)$, and propagation of these results to other blocks underneath $(i+1...n, j+1...n)$. The first of these three tasks is executed sequentially while the other two are executed in parallel. One task is spawned per diagonal block, which in turn, executes 3 tasks within, making a total of $3n$ – 2 tasks (where n is the number of diagonal blocks). The parallelism in tasks (i.e., the number of subtasks) reduces as we move down the matrix because the number of blocks to be solved in parallel decreases. The CPU subtask implementation uses single-precision BLAS functions while the GPU implementation employs the corresponding CUBLAS routines.

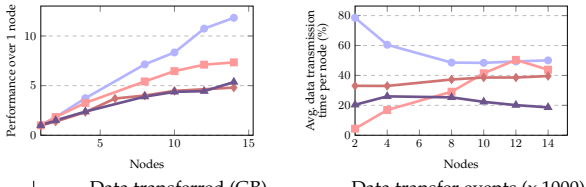
The 2D-FFT experiment performs two single-precision one dimensional complex-to-complex FFTs (one along matrix rows and the other along matrix columns) over a matrix with 61440×61440 elements. The input matrix is initially randomly distributed over the cluster nodes in blocks of 512 consecutive matrix rows. We use two *Unicorn* tasks for the experiment (each with 120 subtasks). The first task performs 1D-FFT along matrix rows while the second performs 1D-FFT along matrix columns. Note that we do not need to perform an explicit transpose in between the two and instead transpose the subscription and rely on the network subsystem. For the first 1D-FFT, the subtask size is 512 rows while it is 512 columns for the second. The CPU subtask callback uses the FFTW [25] library, whereas the GPU subtask uses calls the CUFFT [26] library functions.

The PageRank experiment computes the search rank of a webpage based on the web's hyperlink structure. The search rank of a webpage is the probability of a random surfer visiting it. The algorithm works by first uniformly initializing the ranks of each page to a constant value, and then iteratively transferring the ranks of all web pages to their outlinks till the ranks of all pages converge (or up to a maximum number of iterations).

For our experiment, we use a randomly generated web graph of 500 million web pages and a maximum of 20 outlinks per web page. With 90% probability, a page's outlinks point to nearby pages (within an imaginary circle centered at this web page and having a diameter of 0.1% of total web pages), with 9% probability, a page's outlinks point within a diameter of 1%, and with 1% probability they are linked to farther off pages. The data for the web (along with its outlinks) is stored on NFS in multiple files, with each file storing the data of one million web pages. *Unicorn* memory maps these files on each cluster node, from where all subtasks subscribe.

Our PageRank implementation performs 25 iterations and each iteration executes 250 subtasks. Each subtask processes 2 million distinct web pages and transfers each web page's rank to all its outlinks equally. Note that a web page may point to any other page in the web. Thus, the corresponding subtask may produce output page rank for any web page. For this reason, all subtasks write-subscribe to the entire output address space and the output of all subtasks are summed up (using *Unicorn's* data reduction callback) to produce the final output. The output of subtasks is written to an address space in every iteration and after reduction it becomes the input for the next iteration. We allocate two address spaces and alternate them as input or output every iteration. After the last iteration, the output address space contains the final page ranks. The CUDA kernel code for this experiment uses the GPU's global memory to accumulate page ranks produced by the GPU threads employed by the subtask and synchronization between them is done using global memory atomic add instructions.

5.2 Performance Scaling



Nodes	Data transferred (GB) 2	4	8	10	12	14	Data transfer events (x 1000) 2	4	8	10	12	14
Image Convolution	6.1	9.3	10.6	10.7	11.0	11.4	1.55	3.2	4.46	4.97	5.12	5.38
Matrix Multiplication	17.0	61.5	140.0	186.5	224.0	248.5	0.07	0.21	0.48	0.54	0.67	0.66
LU Decomposition	40.0	79.2	142.9	172.0	198.0	222.5	7.53	15.3	25.4	30.3	32.1	36.7
2D FFT	24.7	45.7	50.3	49.5	53.0	51.2	1.83	3.97	11.18	10.84	10.93	14.58

Fig. 4: Performance analysis of various benchmarks

Results in Figure 4 show strong scaling achieved by our implementations of Image convolution, Matrix multiplication, LU decomposition and 2D FFT. Of the four experiments, Image convolution exhibits the maximum scaling and peaks at 11.83 when run on 14 nodes. Matrix multiplication, being relatively expensive on communication, achieves a maximum scaling of 7.34. In contrast to image convolution, which spends 50.05% of the total experimental time in communication (of 11.4 GB), matrix multiplication transfers a much larger 248.50 GB data in the cluster and this takes 43.81% of the total experimental time. Although for matrix multiplication both input matrices are 16 GB each, every input block is required by all subtasks in its row and all in its column. Thus about 250 GB data is transferred in 659 pipelined events (each of the 1024 subtasks subscribe to 1 GB data from both the input matrices).

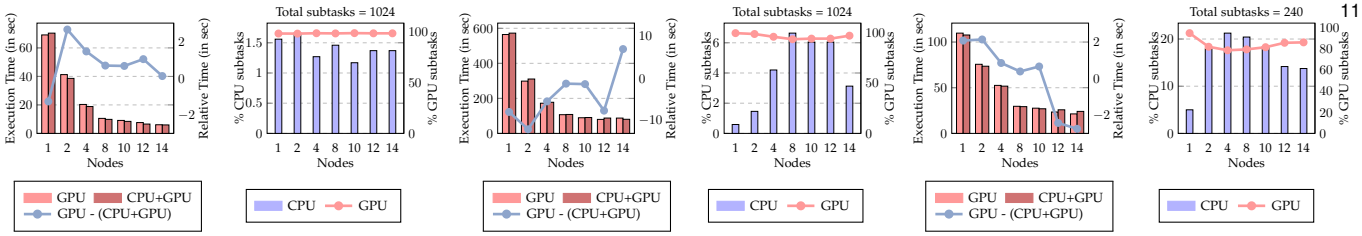


Fig. 5: Image Convolution – GPU vs. CPU+GPU

Fig. 6: Matrix Multiplication – GPU vs. CPU+GPU

Fig. 7: 2D FFT – GPU vs. CPU+GPU

The other two experiments, block LU decomposition and 2D-FFT, exhibit relatively inferior scaling (4.8 and 5.36 respectively) as these experiments have limited parallelism. The former is an iterative experiment with 32 iterations and 3 tasks per iteration (one out of these three is sequential). The experiment has an average of 121.7 subtasks per task. The 2D-FFT experiment (with an implicit matrix transpose operation) has two tasks each with 120 subtasks.

The results in Figure 4 indicate a similar amount of data transfer for matrix multiplication and LU decomposition. However, the former is implemented as a single task with time complexity $O(n^3)$ and the latter has many iterations with three tasks using BLAS calls of varying time complexities ($O(n)$, $O(n^2)$ and $O(n^3)$). This increases communication latency, which is evident from fifty times more data transfer events (36654 *versus* 659). This, coupled with the fact that the first of these three tasks is sequential, leads to lower scalability for the experiment. In contrast, both image convolution and 2D FFT have lower time complexities – $O(nm)$ for the former (m being the filter size) and $O(n \log n)$ for the latter, but the latter has around 4.5x more data transfer, resulting in its lower scaling. These results collectively indicate that applications with high compute to communication ratio should perform well with our runtime.

Figures 5, 6 and 7 plot GPUs-only performances for image convolution, matrix multiplication and 2D FFT respectively and compare those to the corresponding CPU+GPU performances. For the computation in these benchmarks there is a significant performance disparity between CPUs and GPUs, with GPUs being an order of magnitude faster than CPUs. Due to this, employing both together generally results in nearly all CPU subtasks getting *multi-assigned* to GPUs. Also, running some subtasks on slower CPUs takes away the opportunity to pipeline those in case the GPUs had executed them. For these reasons, the experiments' performances when using CPU+GPU do not show much throughput gain over their GPUs-only versions. For image convolution and matrix multiplication, results show that nearly 98% subtasks were executed by GPUs and CPUs were only able to complete fewer than 2% of the subtasks. For 2D FFT, this ratio is a little better with around 15% subtasks completed by CPUs.

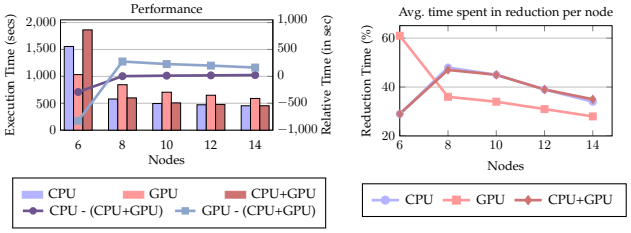


Fig. 8: PageRank

Figure 8 presents results of the PageRank experiment. In the map stage, 250 subtasks process an equal share of input web pages and distribute their page ranks equally among all their outlinks. In the reduce stage, the page rank contributions from each subtask are summed up to compute the final page ranks of all web pages. Reductions are performed in parallel on all cluster nodes. Once the intra-node reductions complete, nodes perform inter-node reductions to compute the final result.

Results show that CPUs perform better than GPUs for this experiment. This is because of the large overhead of atomic-add operations on GPUs. Further, there is an additional overhead of large GPU↔CPU data transfers. Along with performance results, Figure 8 also indicates that the average time spent in the reduction stage decreases with an increasing number of nodes. This is because more nodes allow more reductions to be simultaneously executed.

5.3 Work Stealing

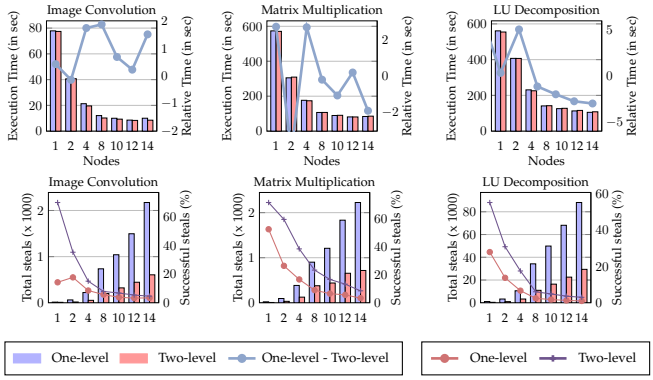


Fig. 9: One-level versus two-level work stealing

Next, we present a comparison of one-level work stealing with *Unicorn's* two-level scheme with a *steal agent* running per node (section 4.3.2). Results in Figure 9 show comparable performance numbers for one-level and two-level scheduling schemes for matrix multiplication and LU decomposition. However, image convolution reports 6.92% average performance improvement with two-level stealing as compared to the one-level stealing scheme. This is because the small memory footprint of an image convolution subtask allows GPUs (which are the stealers in most cases) to co-execute more subtasks as compared to other experiments where despite stealing a set of subtasks large memory footprint prohibits their co-execution.

Despite moderate performance gains, two-level stealing reports significant reduction (over one-level) in the total number of steal attempts generated in the cluster. The bottom half of Figure 9 plots total steal attempts in the cluster with bars while the successful steals are plotted with lines. The average reduction (in the number of total steal attempts) for image convolution, matrix multiplication and LU decomposition respectively is 65.88%, 64.9% and 67.03%. Due to this, the average number of successful

steals in the cluster increase in two-level scheme (over one-level) by 44.24%, 54.25% and 60.08% for these experiments respectively. The same is not translated into performance gains because our steals are extremely light weight. The only information transferred as a result of steal is the *subtask ID* and there is no associated direct data transfer.

5.4 Load Balancing

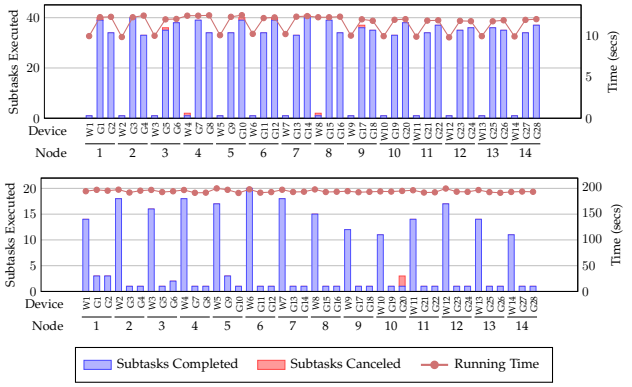


Fig. 10: Load Balancing (Top: Image Convolution, Bottom: PageRank) – W is a CPU work group and G is a GPU device

In this section, we study the effectiveness of our scheduler in achieving a balanced load on all cluster devices. Figure 10 plots the finishing times of all cluster devices for the image convolution benchmark and for one iteration of the PageRank experiment. The figure also plots the number of subtasks executed by each of these devices. Note that all the CPU devices on a node are represented as work-groups numbered from $W1$ to $W14$. Similarly, GPU devices in the cluster are labelled $G1$ to $G28$.

Recall that for the matrix multiplication experiment, the entire input data is initially equally distributed randomly among all cluster nodes. For this reason, all GPUs execute roughly the same number of subtasks (as they face similar data transfer overheads and subtasks are homogeneous). The same is true for CPU work-groups. For the PageRank experiment, however, the input data is resident on NFS. The graph plots 10th iteration of the experiment which means that the input data for the iteration additionally comes from different cluster nodes (as it is computed in last iteration). As such, variable input data latency is incurred by various CPU work-groups, causing them to execute different number of subtasks. Despite the disparity in subtask execution rate of GPUs and CPU work-groups, the finishing times of each of them is quite close to each other (for both experiments). This shows that our scheduler is able to balance the cluster load despite this device heterogeneity.

5.5 Stress Tests

In this section, we put our runtime under non-favorable conditions and study its response to various experiments.

5.5.1 Heterogeneous Subtasks

Figure 11 shows load balancing achieved by *Unicorn* when subtasks in Image Convolution experiment perform different amount of work. The top half of the image is convolved using 512 subtasks of size 2048×2048 while the bottom half is convolved using 128 subtasks of size 4096×4096. The experiment is executed on 10 nodes. Despite, the four fold execution disparity in subtasks, our runtime maintains a decent load balance in the cluster.

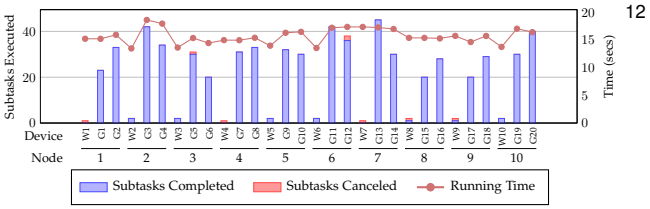


Fig. 11: Load Balancing (Heterogeneous Subtasks) – W denotes a CPU work group and G denotes a GPU device

5.5.2 Varying Subtask Size

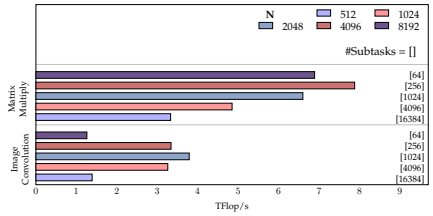


Fig. 12: Subtask size ($N \times N$)

Figure 12 shows the response of our runtime to choices of the user in sizing subtasks. Within a reasonable range – (2048-8192) for matrix multiplication and (1024-4096) for image convolution – our system is able to maintain a throughput within 20% of the peak performance. On either side, system overheads begin to dominate. For extreme sizes, the throughput degrades as on one extreme there are too few subtasks to generate enough parallelism and on the other there are too many subtasks resulting in data transfers dominating the exploitable parallelism.

5.5.3 Input Data Distributions

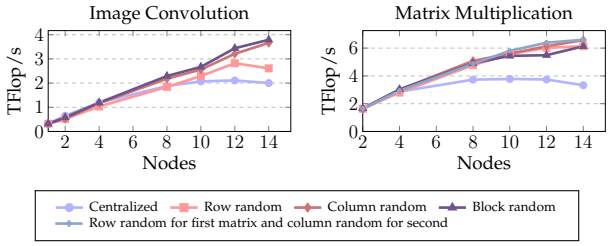


Fig. 13: Impact of initial data distribution pattern

Figure 13 evaluates image convolution and matrix multiplication for various initial input data placement schemes (in the address spaces) like *centralized* (entire address space on one cluster node), *row random* (rows of 2048×2048 blocks spread randomly on all nodes), *column random* (columns of 2048×2048 blocks placed randomly on all nodes) and *block random* (2048×2048 blocks placed randomly on all nodes). Additionally, a fifth scheme is plotted for matrix multiplication where rows of 2048×2048 blocks for the first input matrix and columns of 2048×2048 blocks for the second input matrix are placed randomly in the cluster. Results show that our runtime maintains performance across the various data availability pattern. Only the *centralized* scheme behaves poorly as the network interface of the node containing the entire data becomes a bottleneck.

5.6 Unicorn Optimizations

In this section, we study the impact of a few *Unicorn* optimizations.

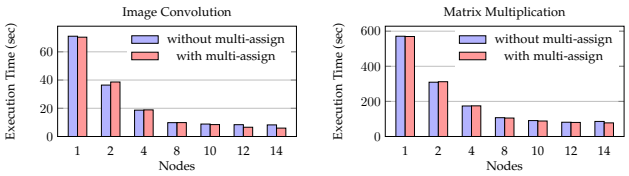


Fig. 14: Multi-Assign (no external load)

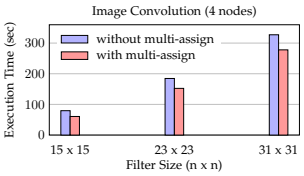


Fig. 15: Multi-assign (external load)

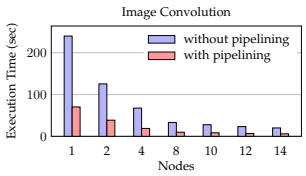


Fig. 16: Pipelining

5.6.1 Multi-Assign

We study two cases to understand the implications of *multi-assign*. First, in the absence of any external load we compare the performance of image convolution and matrix multiplication experiments with *multi-assign* enabled to the case with *multi-assign* disabled. Second, we create an external load and study how image convolution behaves with and without *multi-assign*.

For the first case, results in Figure 14 show that there is no performance penalty in enabling *multi-assign*, in general. The observed average performance gain (with *multi-assign*) for image convolution and matrix multiplication respectively are 9.19% and 2.23% respectively (in comparison to no *multi-assign*). In fact, the performance gains increase with an increasing number of nodes – for image convolution, *multi-assign* reports a maximum gain of 38.26% (over no *multi-assign*) in the fourteen node case. Similarly, the maximum performance gain of 10.62% is observed in the fourteen node case for matrix multiplication.

Next, we study multi-assign over four nodes with the image convolution benchmark. We artificially overload one of the nodes with one process per core computing trigonometric functions indefinitely. In this case, we expect subtasks assigned to the overloaded node to be moved away from it. Our scheduler does this through stealing and multi-assign. In the absence of multi-assign a subtask may start running on a slow device and take a long time to finish, thereby delaying the task. We run this test twice: once allowing multi-assign and once preventing it. Results in Figure 15 show that with multi-assign, subtasks of the overloaded cores get re-assigned and the task completes faster. Without it, the task remains bottlenecked by the 'slow' cores. Our heuristics generally only multi-assign fewer than 1% of the subtasks, but the later-assigned unit finishes first about 50% of the time. Of course, when one finishes, the other is aborted, leading to a faster overall time. The cancellation protocol itself has insignificant overhead.

5.6.2 Pipelining

Figure 16 shows the impact of pipelining with the help of the image convolution experiment. With pipelining, our runtime overlaps computation of one subtask with the communication of the next. Further, on GPUs this makes multiple simultaneous kernel executions possible. Results show that our runtime achieves 3-4x speed-up with pipelining.

5.6.3 Software cache for GPUs

Unicorn employs a software cache to reduce DMA (Direct Memory Access by GPU without involving CPU) data transfers to all GPUs in the cluster. The cache prevents read-only data shared by two or more subtasks (of a task) executing on a GPU from being DMA'ed more than once. In this section, we study five cache eviction policies and compare their performances. The five policies are LRU: *least recently used*, MRU: *most recently used*, LFU: *least frequently used*, MFU: *most frequently used* and Random. Figure 17 plots the performances of these policies for the matrix multiplication experiment along with the scenario when no GPU cache is used. Results show that both LRU and LFU perform better than others. This result can also be inferred from fewer cache evictions per allocation and fewer average cache misses in LRU and LFU, compared to MRU, MFU and Random.

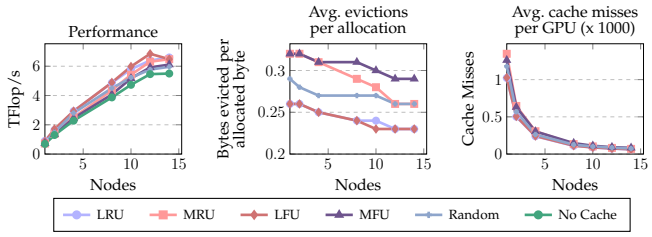


Fig. 17: Cache Eviction Strategies (Matrix Multiplication)

5.6.4 Data Compression

Nodes	Network reduction statistics			GPU→CPU reduction statistics		
	Uncompressed Data Size (GB)	Compression Ratio	Performance Gain (%)	Uncompressed Data Size (GB)	Compression Ratio	Performance Gain (%)
8	162.98	3.63	6.44	27.01	44.63	0.31
10	209.55	4.72	16.10	25.15	44.64	0.83
12	256.11	5.29	21.00	23.28	44.66	0.64
14	302.68	5.78	23.81	31.67	44.61	2.45

Fig. 18: PageRank data compression (250 million web pages)

To ameliorate high data transfer latency in PageRank reduction, we compress data computed by subtasks before transferring over the network or from GPU to CPU. The employed compression algorithm is based on Run Length Encoding (section 4.6) and is executed on GPUs for GPU→CPU data transfers and on CPUs for inter-node data transfers. Figure 18 presents the cluster-wide size of uncompressed data for reduction, compression ratio and performance boost with compression (over the non-compression case). Results show that performance gains increase with increasing number of nodes and reach a peak of 23.81% for network compression and a peak of 2.45% for GPU→CPU compression (for the 14 node case). Note that because of lower latency of GPU→CPU transfers, the gain observed with GPU compression is moderate as compared to inter-node compression. Also, note that the amount of data transferred in the cluster increases with the number of nodes. However, this reduces the number of subtasks processed per node, which means that inter-node reductions take place with more sparsity in data (leading to higher compression).

5.7 Overhead Analysis

We evaluate *Unicorn's* runtime overhead by varying the number of CPU cores used in experiments and by studying the amount of time experiments spend inside runtime's code versus benchmark application code. We also study the number of times each byte is transferred in the cluster when subtasks have overlapping subscriptions.

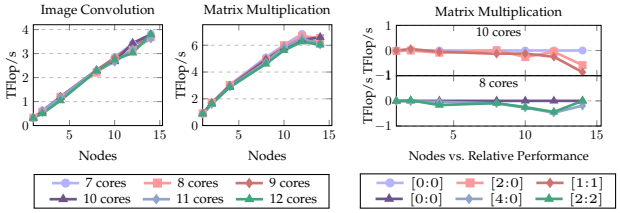


Fig. 19: Varying CPU cores used in subtask computation

5.7.1 Varying CPU cores

CPU cores are not only used for subtask computations but for many other critical operations like CPU-GPU data transfers, network data transfers, scheduling and *Unicorn* runtime's control operations. For all our experiments, we have reserved two CPU cores (out of 12 available) per node for these support functions and presented results by using the rest for subtask computations. Figure 19 varies the number of CPU cores allowed for the application subtasks per node from 7 to 12 and compares their performances. Results show that the performance of all these cases remain close to each other. The average difference between the maximum and minimum performing cases (at all data points) for both experiments is less than 10%.

This narrow performance difference is an indication of our runtime's low overhead. Thus, we currently do not dynamically vary the number of CPU cores reserved for control operations. An exploration of this is desired in the future. Currently, we only allow applications to statically specify the number of CPU cores to be reserved.

We also study the impact of binding our *compute threads* (section 4.1) to CPU cores. In all the experiments presented thus far, we have not explicitly bound *compute threads* to processing cores, allowing the operating system to manage them. In the rightmost graph of Figure 19, we compare this scenario to the case when all our *compute threads* are explicitly bound to CPU cores (which leaves a few cores for other critical operations like data transfers). The figure plots the performance of the matrix multiplication experiment for two cases - when 10 cores and 8 cores are employed in subtask computation. For each case, we report performance when there is no explicit core for these critical operations (plotted as $[0{:}0]$). Relative to this, we plot performances of cases where $[n{:}m]$ cores are designated for non-subtask computations (i.e. n cores are explicitly freed from first CPU while m cores are explicitly freed from the second one). Results show that explicit binding of *compute threads* (to cores) does not have a significant performance impact. However, not explicitly binding threads and letting the operating system to freely migrate them yields best performance.

5.7.2 Unicorn Time versus Application Time

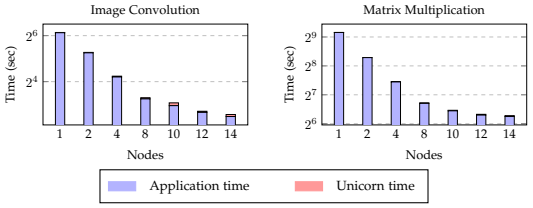


Fig. 20: Library time versus application time

Figure 20 compares the time spent (by image convolution and matrix multiplication experiments) in runtime's code to the time spent in application execution. The latter includes the time taken for data transfers (both network and CPU-GPU) and callback executions. The rest of the experimental time is considered as our runtime's overhead. Results show that the average overhead for image convolution is 3.6% while it is 0.91% for matrix multiplication. In absolute value, the average overhead is 0.43 seconds for image convolution and 0.82 seconds for matrix multiplication.

5.7.3 Data Transfer Frequency

Nodes	Image Convolution			Matrix Multiplication			
	Avg. unique data sent per node (GB)	Avg. total data sent per node (GB)	Avg. transfer freq. per byte per node	Avg. unique data sent per node (GB)	Avg. total data sent per node (GB)	Avg. transfer freq. per byte per node	Avg. first matrix transfer freq. per byte per node
2	3.05	3.05	1.00	8.50	8.50	1.00	1.00
4	2.26	2.27	1.00	7.38	15.88	2.15	1.18
8	1.31	1.32	1.01	5.63	18.19	3.23	1.86
10	1.07	1.07	1.01	4.50	18.20	4.04	2.17
12	0.91	0.92	1.01	4.00	18.83	4.71	2.78
14	0.87	0.88	1.01	3.18	18.36	5.78	2.88

Fig. 21: Data Transfer Frequency

In this section, we study the number of times each byte in the address space gets transferred in the cluster (between nodes). Two examples are considered – on one end is the image convolution experiment which has very little subscription overlap (fringe) among subtasks. At the other end is matrix multiplication where entire input data of every subtask overlaps with that of other subtasks. Figure 21 lists the average number of unique and total bytes transferred by every node in the cluster along with the average data transfer frequency (i.e., the ratio of total bytes transferred to unique bytes transferred per node). Results show that the data transfer frequency stays close to 1 for the image convolution experiment, whereas it grows with increasing number of nodes for the matrix multiplication experiment. This is because one of the input matrices is required on all cluster nodes and because of the initial random placement of data, $(n-1)$ transfers are required (where n is the number of nodes). The other matrix, however, exhibits relatively moderate transfer frequency (the last column in the table). However, most of the data transferred is only because it resides at a node different from the subtask that requires it. For matrix multiplication, e.g., there is only 3.88% additional transfer due to re-sending of data to the new destination after stealing or multi-assignment.

5.8 Unicorn versus others

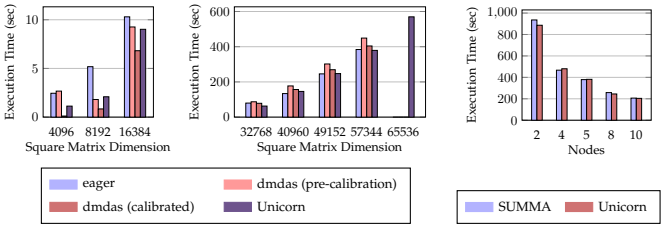


Fig. 22: Unicorn versus StarPU and SUMMA

In this section, we compare our runtime to two other parallel computing systems – StarPU [27] (a single node framework) and SUMMA [28] (a hand tuned multi-node benchmark for matrix multiplication). The left and middle graphs in figure 22 compare the performance of our matrix multiplication implementation to StarPU. The three bars for StarPU plot, respectively, its default *eager* scheduler, the first run of its advanced *dmdas* scheduler (i.e., without calibration; this scheduler requires calibration runs for better performance), and the best run out of three successive runs

of *dmdas* after calibration. This best run performs better than *Unicorn* until a matrix size of 16384×16384. For larger matrices, it starts to lag Unicorn and eventually fails at 65536×65536 reporting it ran out of memory. Yet, *Unicorn* runs at this and even higher sizes. The rightmost graph in the figure compares *Unicorn* to SUMMA for multiplication of two square matrices (on CPUs only) of size 32768×32768. Results show that *Unicorn* performs quite close to SUMMA for this double precision computation. Note that SUMMA incurs a bit of overhead in the sense that it requires a different MPI process per CPU core whereas *Unicorn* works with one MPI process per node. For this experiment, we have used different block sizes for both implementations (1024×1024 for *Unicorn* and 128×128 for SUMMA) in order to compare their best CPU performances.

6 Related Work

Among others, StarPU [27], XKappi [29] and MAPS-Multi [30] are a few notable examples of comprehensive parallel programming frameworks that provide some form of scheduling and load balancing on single node systems. While the first two support both CPUs and GPUs, MAPS-Multi [30] targets only multiple GPUs on a node and studies workload distribution on the basis of data access patterns. Existing cluster programming systems can be broadly classified into two categories. The first comprises language based approaches like UPC [31], Split-C [32], Cilk [33] and Co-array Fortran [34]. These usually extend a sequential language like C or Fortran. The second category includes library based approaches like Globus [35], MPI-ACC [36], PVM [37] and BSPlib [38]. The former ones focus variously on functional, loop or data parallelism and generally use shared address spaces (built on top of DSM or more specifically PGAS [39]) with fine grained synchronization. Their focus is to mainly allow the user to express parallelism at a high level and most do not support GPUs. In contrast, library based approaches employ some MPI-like communication, where machine specific details are not completely abstracted from the programmer. Instead of focusing on program logic, the programmer has to directly handle issues like synchronization, scalability and latency. Hence, usual problems like race conditions and deadlocks remain.

Our system *Unicorn* is built on top of pthreads, MPI, CUDA and OpenCL. Its novelty is in its general and intuitive interface, yet efficient implementation. It unifies computation on local and remote computing units (CPUs and GPUs) using bulk synchrony. Its runtime environment autonomously performs data distribution, dynamic load-balanced scheduling and synchronization. The closest existing works targeting CPU and GPU clusters are StarPU-MPI [2], G-Charm [3] and Legion [4]. StarPU-MPI is an extension of StarPU but it does not fully abstract the existence of multiple machines: the programmer must either explicitly manage communication with an MPI-like interface or explicitly submit independent tasks to each node of the cluster. Rather than a unified cluster programming framework, it is an MPI based aggregation of independent StarPU instances running on each node. It lacks a cluster wide scheduler. It only provides independent schedulers on each node, with inter-node schedule managed by the user. StarPU maintains data replicas for potential use in upcoming tasks. However, if a task modifies data in one of the replicas, all others are invalidated. For good performance, it recommends applications to advise when and where not to keep data replicas. In contrast to this, *Unicorn* adopts a light weight memory consistency protocol where the invalidation messages are deferred to task boundaries (where they are piggy-backed on other regular message exchange between nodes) and no overhead is incurred during task execution. StarPU also supports a notion of data filters which allows data to be viewed in parts (or hierarchy) by associated codelets. These filters are usually synchronous. Asynchronous filters can also be created with some limitations on data usage by the application. *Unicorn*, on the other hand, is an entirely asynchronous system and achieves data partitioning through an application callback.

G-Charm, based on Charm++ [40], is a framework specially optimized for GPUs. It particularly focuses on reducing GPU data transfers by employing a software-cache over GPUs and grouping multiple Charm++ *chares* together to reduce the number of GPU kernel invocations. Each processor in the system runs an independent instance of Charm++ runtime and they communicate via message-passing (messages are buffered in a message queue). Even the input data for chares is received by this mechanism. In contrast, *Unicorn* has a dedicated thread (on each node) to efficiently manage data transfers and segregate them from control messages. This approach guarantees progress of the entire asynchronous system. Also, unlike *Unicorn's multi-assign*, G-Charm lacks a mechanism to reconsider and correct poorly made scheduling decisions.

Legion is a powerful system that uses a software out-of-order processor to schedule application-created tasks. Legion runtime requires programmers to select where tasks run and how data regions are placed. The runtime follows a deferred execution model where events (may be programmer specified) define task graphs and control task execution. *Unicorn* runtime, on the other hand, leaves lesser controls with applications and manages data placement and task graphs by itself.

7 Conclusions and Future Work

We present a practical bulk synchronous programming model that allows distribution of computation across multiple CPU cores within a node, multiple GPUs, and multiple nodes connected over a network, all in a unified manner. Our model maps efficiently to modern devices like GPUs, as they are already bulk synchronous in nature. Our runtime undertakes all local and networked data transfers, scheduling, and synchronization in an efficient and robust manner. By design, we eliminate races and deadlocks as all devices operate in a private view of the address space.

Unicorn exposes a distributed shared memory system with transactional semantics. Experiments show that our runtime overcomes the traditional performance limitations of the approach and achieves good performance gains. This is enabled by a number of critical optimizations working in concert. These include prefetching, pipelining, maximizing overlap between computation and communication, and scheduling/re-scheduling efficiently across heterogeneous devices of vastly different capacities. *Unicorn* also employs special optimizations for GPUs like a software LRU cache

to reduce DMA transfers and a proactive work-stealer to reduce pipeline stalls. Our framework can realize any task that may be optionally decomposed into a set of concurrently executable subtasks with checkout/checkin memory semantics and a synchronized reduction step to resolve conflicting checkins. However, tasks having non-deterministic access pattern (like graph traversal) or fine-grained/frequent communication or complex conflict resolution may not perform efficiently in our system.

In the future, there is potential to optimize data transfers for a set of tasks rather than one task at a time. Except for data dependency or an explicitly specified user dependency, *Unicorn* tasks are independent of each other. However, wiser scheduling decisions can be made with à priori knowledge of tasks to come. *Unicorn's* locality aware scheduler produces a schedule optimized for the task at hand. By evaluating the data requirements of dependent tasks, it is possible to produce a globally optimal schedule. Of course, the time it takes to generate such a schedule must be weighed against the time saved in data transfers while executing the global schedule.

REFERENCES

[1] Beri *et al.*, "A scheduling and runtime framework for a cluster of heterogeneous machines with multiple accelerators," in *Parallel and Distributed Processing Symposium (IPDPS)*, 2015.

[2] Augonnet *et al.*, "StarPU-MPI: Task programming over clusters of machines enhanced with accelerators," in *Euro. Conf. Recent Advances in the MPI*, ser. EuroMPI, 2012.

[3] Vasudevan *et al.*, "G-charm: An adaptive runtime system for message-driven parallel applications on hybrid systems," in *International Conference on Supercomputing*, ser. ICS '13, 2013.

[4] Bauer *et al.*, "Legion: Expressing locality and independence with logical regions," in *International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12, 2012.

[5] Min *et al.*, "Supporting realistic OpenMP applications on a commodity cluster of workstations," in *Intl. Conf. on OpenMP Shared Memory Parallel Programming*, ser. WOMPAT, 2003.

[6] Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, 1990.

[7] Dean and Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, Jan. 2008.

[8] Page *et al.*, "The PageRank citation ranking: Bringing order to the web." Stanford InfoLab, Technical Report 1999-66, Nov. 1999.

[9] Gropp *et al.*, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Comput.*, vol. 22, no. 6, Sep. 1996.

[10] Guttman, "R-trees: A dynamic index structure for spatial searching," *SIGMOD Rec.*, vol. 14, no. 2, Jun. 1984.

[11] Lifflander *et al.*, "Work stealing and persistence-based load balancers for iterative overdecomposed applications," in *Intl. Symp. High-Perf. Parll. and Dist. Comput.*, ser. HPDC, 2012.

[12] Dinan *et al.*, "Scalable work stealing," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC, 2009.

[13] Dongarra *et al.*, "An extended set of FORTRAN basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 14, no. 1, Mar. 1988.

[14] "The NVIDIA CUDA basic linear algebra subroutines," https://developer.nvidia.com/cuBLAS.

[15] Stone *et al.*, "OpenCL: A parallel prog. standard for heterogeneous comput. sys." *IEEE Des. Test*, vol. 12, no. 3, 2010.

[16] Nickolls *et al.*, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, Mar. 2008.

[17] Min *et al.*, "Hierarchical work stealing on manycore clusters," in *Partitioned Global Address Space Programming Models*, 2011.

[18] Beri *et al.*, "Prosteal: A proactive work stealer for bulk synchronous tasks distributed on a cluster of heterogeneous machines with multiple accelerators," in *IPDPS 2015*.

[19] Li and Mascagni, "Improving performance via computational replication on a large-scale computational grid." in *CCGRID*, vol. 3.

[20] Nibhanupudi and Szymanski, "Adaptive parallelism in the bulk-synchronous parallel model," in *European Conference on Parallel Processing*.

[21] Beri *et al.*, "Locality Aware Work-Stealing based Scheduling in Hybrid CPU-GPU Clusters," in *PDPTA*, 2015.

[22] Gabriel *et al.*, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Euro. PVM/MPI Users Group Meeting*, 2004.

[23] InfiniBand Trade Association, InfiniBand Architecture Specification, Release 1.1, 2002.

[24] Demmel *et al.*, "Block LU factorization," http://www.netlib.org/utk/papers/factor/node7.html, 1995.

[25] Frigo and Johnson, "FFTW: an adaptive software architecture for the FFT," in *ICASSP*, 1998.

[26] "CUFFT: CUDA fast fourier transform (FFT)," http://docs.nvidia.com/cuda/cufft/.

[27] Augonnet *et al.*, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurr. Comput. : Pract. Exper.*, vol. 23, no. 2, pp. 187–198, Feb. 2011.

[28] Geijn and Watts, "SUMMA: Scalable universal matrix multiplication algorithm," Austin, TX, USA, Tech. Rep., 1995.

[29] Gautier *et al.*, "XKaapi: A runtime system for data-flow task programming on heterogeneous architectures," in *IPDPS '13*.

[30] Ben-Nun *et al.*, "Memory access patterns: The missing piece of the multi-GPU puzzle," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15, 2015.

[31] El-Ghazawi and Smith, "UPC: Unified parallel C," in *ACM/IEEE Conf. Supercomputing*, ser. SC, 2006.

[32] Krishnamurthy *et al.*, "Parallel programming in Split-C," in *ACM/IEEE Conf. Supercomputing*, ser. SC, 1993.

[33] Blumofe *et al.*, "Cilk: An efficient multithreaded runtime system," *SIGPLAN Not.*, vol. 30, no. 8, Aug. 1995.

[34] Numrich and Reid, "Co-array Fortran for parallel programming," *SIGPLAN Fortran Forum*, vol. 17, no. 2, Aug. 1998.

[35] Foster, "Globus toolkit version 4: Software for service-oriented systems," in *Intl. Conf. on Network and Parll. Comput. 2005*.

[36] Aji *et al.*, "MPI-ACC: An integrated and extensible approach to data movement in accelerator-based systems," in *HPCC'12*.

[37] Beguelin *et al.*, "A user's guide to PVM parallel virtual machine," University of Tennessee, Tech. Rep., 1991.

[38] Hill *et al.*, "BSPlib: The BSP programming library," *Parallel Comput.*, vol. 24, no. 14, 1998.

[39] Stitt, "An introduction to the partitioned global address space (PGAS) programming model," in *Connexions*, Mar 2010.

[40] Kale and Krishnan, "Charm++: A portable concurrent object oriented system based on C++," Tech. Rep., 1993.

Tarun Beri received the BTech degree in computer science from Thapar University, India and an MS degree from IIT Delhi. He is currently working towards the PhD degree in the Department of Computer Science at IIT Delhi. His interests include high performance computing, GPGPU, machine learning and vector graphics.

Sorav Bansal is a faculty member at the CSE department in IIT Delhi, and works in the areas of programming languages and operating systems. His research efforts have been around redefining OS abstractions for modern heterogeneous computing environments, investigating superoptimization-based compiler design patterns, and improving virtualization performance. Sorav obtained his B.Tech. from IIT Delhi, and M.S. and Ph.D. from Stanford University.

Subodh Kumar is a faculty member in the department of Computer Science at IIT Delhi. He was earlier a faculty member at Johns Hopkins University. Prof. Kumar received his bachelor degree in computer science and engineering from IIT Delhi, and an MS and a PhD in computer science from The University of North Carolina at Chapel Hill. His primary areas of interest include computer graphics, geometry processing, virtual worlds, GPGPU and parallel computation.