Locality Aware Work-Stealing based Scheduling in Hybrid CPU-GPU Clusters

A. Tarun Beri[1], **B. Sorav Bansal**[1], **and C. Subodh Kumar**[1]
[1]Indian Institute of Technology Delhi, New Delhi, India
{tarun,sbansal,subodh}@cse.iitd.ac.in

Abstract— *We study work-stealing based scheduling on a cluster of nodes with CPUs and GPUs. In particular, we evaluate locality aware scheduling in the context of distributed shared memory style programming, where the user is oblivious to data placement. Our runtime maintains a distributed map of data resident on various nodes and uses it to estimate the affinity of work to different nodes to guide scheduling. We propose heuristics for incorporating locality in the stealing decision and compare its performance with a locality oblivious scheduler. In particular, we explore two heuristics that focus on minimizing the cost of fetching data that is non-local. These heuristics respectively minimize the number of remote data transfer events, and the number of remote virtual memory pages fetched. Finally, we also study the impact of different placements of the initial input, like block cyclic, random and centralized, on the scheduler.*

We implement and evaluate these schedulers within Unicorn, a heterogenous framework that decomposes bulk synchronous computations over a cluster of nodes. Compared to a locality oblivious scheduler, the average observed overhead of our techniques is less than 8%. We show that even with this overhead, average performance gain is between 10.35% and 10.6% in LU decomposition of a one billion element matrix and between 12.74% and 14.55% in multiplication of two square matrices of one billion elements each on a 10-node cluster with 120 CPUs and 20 GPUs.

Keywords: Locality aware, Work stealing, Hybrid CPU-GPU clusters, Distributed computing

1. Introduction

Distributed shared memory based frameworks like Unicorn [1], Global Arrays [2] and X10 [3] allow programming styles simpler than message passing. User only accesses "memory" and the underlying data packing and communication is managed transparently by the runtime. Further, the computational tasks are also scheduled and load-balanced by the runtime. In this paper, we specifically aim to reduce the time for which computation is blocked behind network latency induced by remote memory access. We do this by scheduling data transfer early, overlapping it with other computation. Further, a number of heuristics are proposed to schedule computation close to data. In particular, an affinity is computed in a distributed fashion from partial information available at each node and input to a greedy scheduler. We develop these optimizations within Unicorn [1], a parallel programming framework for clusters populated with both CPUs and accelerators like GPUs.

Traditionally, locality-aware scheduling has centered on cache-affinity and focus has been on improving cache reuse. However, we argue that *node-affinity* is equally important for hybrid clusters, especially because significant time can be lost in fetching remote data and different devices are able to consume data at different rates. GPUs, being computationally more aggressive, cause more performance loss than CPUs if they need to wait for remote data transfer. For maintaining optimal data throughput (for achieving peak performance), avoiding GPU stalls on data is important and having à priori knowledge of data locality is useful, which Unicorn provides. In distributed environments, node affinity is often left to the user to specify [3]. In this paper, we instead explore inferring affinity based on the data accessed by the computation. We then inform the scheduling algorithm with this affinity value.

Unicorn [1] decomposes user's tasks into many independent *subtasks*. Subtasks are concurrent, and dependencies are only between tasks. Unicorn transparently schedules each spawned subtask to execute on an available computing device in the cluster. Information about data that a subtask seeks to use is specified early. Establishing subtask to node affinity based on this anticipated usage then can guide preferential scheduling of subtasks on nodes where most of its required data is resident. This, however, may not be optimal. Scheduling computation close to the largest resident data does not consider the time required to transfer the remaining data, which may offset the transfer time that is saved. For instance, the remote data may be highly dispersed among other nodes. Hence, we present other approaches, which instead of merely maximizing data locality focus on minimizing the transfer time of non-local data.

Note that all computations of affinity require an analysis of the set of addresses accessed by each subtask. This information, even if statically provided, is too large to store and process and reductions are necessary. This is the subject of this paper.

Many parallel and distributed programming frameworks, including Unicorn, employ randomized work-stealing for load balancing. Data locality of random work-stealing has been extensively studied for shared memory programming

and it has been largely found to be cache-unfriendly [4]. In this paper, we study the data locality of random work-stealing for computations distributed on hybrid CPU-GPU clusters and find that random work-stealing remains node-memory unfriendly as well. We have incorporated locality-aware strategies in work stealing as well to improve this.

As a base case, we start with trying to schedule a subtask on a node where where most of its input resides, maximizing *local data*. As shown in section 4, this technique reports an average performance gain of 12.7% over a non locality-aware work-stealer, while multiplying two square matrices of size $32768 * 32768$.

Next, we experiment with two strategies that instead of maximizing local data, target the time to fetch remote data. Our first strategy minimizes the number of remote data *transfer events* or requests. It is based on the observation that the incurred data fetch latency grows with data fragmentation and the number of data transfer requests. Accessing closely placed remote data is less expensive than accessing discontiguous remote data, which may cost additional latency. We observe an average performance gain of 14.55% with this heuristic over locality oblivious scheduling for matrix multiplication.

Our second strategy directly optimizes for the amount of *remote data*. It minimizes the total number of virtual memory pages to be fetched from remote nodes. Compared to *Unicorn's* locality oblivious scheduling, this strategy reports a performance gain of 12.74% for matrix multiplication.

These strategies behave differently for different experiments. For instance, our block LU factorization experiment respectively reports an average performance improvement (over *Unicorn's* locality oblivious scheduler) of 10.35% for *local data* heuristic, 10.35% for *transfer events* heuristic and 10.61% for *remote data* heuristic. We present more details on these techniques, including their overheads, in section 4.

Finally, we experiment with a few common placements of the initial input. A good scheduling strategy should adapt to the change in input data availability at various nodes in the cluster. We experiment with four different initial data placements - *centralized*, *row cyclic*, *column cyclic* and *random*. In the *centralized* scheme, all input data is placed on one node and all others contain no data initially. In the *row cyclic* scheme, blocks of rows of a pre-defined size are cycled through the cluster nodes in order. The same is done with the columns in the *column cyclic* scheme. In the *random* scheme, 2D blocks of a pre-defined size are kept on randomly selected nodes in the cluster. Results indicate that affinity based heuristics outperform the locality-oblivious scheduler.

The primary contributions of this paper are:

1) We dynamically estimate affinity of computation to nodes based on partial residency information. We show that efficient affinity based scheduling is possible even without full residency information.
2) We evaluate multiple heuristics to compute subtask-node affinity scores, considering data locality and fragmentation.
3) We study the benefit of incorporating affinity in work-stealing in heterogeneous environments. We also evaluate its robustness to different initial data placements.

2. Related Work

Data transfer overheads dominate many parallel applications. Locality aware scheduling is an effective way of reducing data transfers and the vital time spent in communication. Locality awareness has been extensively studied for both CPUs and GPUs. On CPUs, locality aware thread schedulers focus on improving data cache reuse. On GPUs, the focus is on enhancing accelerator kernels by scheduling GPU threads on streaming multiprocessors with better locality.

The locality issue in multi-threaded computations has received a lot of attention in the past. Acar et al. [4] minimize cache invalidations in random work stealing to develop a cache aware work-stealer for a single-core SMP. Similarly, Philbin et al. [5] describe an algorithm that determines a thread execution order that minimizes L2 cache misses. Tam et al. [6] and McGregor et al. [7] group threads based on cache locality for multi-threaded computations on multi-core processors. Vaswani et al. [8] present an analytical model to evaluate the effect of cache affinity on shared memory multi-processing. Intel TBB [9] enhances cache hits by creating an affinity between an iteration and a worker thread, which tends to execute the same iteration over and over. SLAW [10] is an adaptive locality aware scheduler for multi-core SMPs that uses programmer provided locality hints. Huang et al. [11] extends OpenMP [12] for specifying programmer controlled locality that minimizes the cost of data accesses.

Among the GPU locality aware schedulers, Nugteren et al. [13] reorder GPU threads automatically for improving memory coalescing and bank locality. Sugimoto et al. [14] improve cache locality for memory intensive texture-based volume rendering by dynamically varying the width and height of thread blocks so that memory access strides for warps are minimized. Unkule et al. [15] improve memory performance by automatically restructuring GPU kernels to better exploit data locality at the register and shared-memory levels. Lee et al. [16] analyze nested parallel computational patterns (like Map Reduce) for data locality and map them to the target GPU's multi-dimensional thread hierarchy.

FLAME [17] and MAGMA [18] are linear algebra systems that support dynamic scheduling on multiple GPUs. StarPU [19] is another framework with scheduling capabilities on multi-CPU plus multi-GPU architectures but most of its schedulers are locality oblivious. XKaapi [20], however, is a comprehensive runtime system with a locality-aware work-stealing scheduler for a single node application using both multi-core CPUs and many-core GPUs.

Some middleware have also been proposed for improving locality awareness of data intensive applications. SLAM [21]

is one such system that employs a distributed file system and a data-centric scheduler to reduce data transfers while reading from the file system. Similarly, VisDSI [22] proposes a locality aware I/O solution for data visualization.

In this paper, we extend ideas in these systems to build an efficient locality-aware work-stealing scheduler for a cluster of nodes with CPUs and GPUs. We study various scheduling heuristics with the aim of minimizing the time spent in data transfer. Our techniques are implemented in Unicorn parallel programming framework [1], which is briefly described in the next section. We start with a simple *node-affinity* based work-stealing scheduler that maximizes the use of local data and gradually build other heuristics, which focus on minimizing the remote access latency of non-local data.

3. Scheduling

We explore affinity based scheduling within the context of Unicorn [1], a unified parallel programming framework for CPUs and accelerators like GPUs. We first describe Unicorn's scheduling algorithm.

3.1 Unicorn

Unicorn models CPUs and accelerators as bulk synchronous computing devices that operate in logically distinct phases of local computation and synchronization. An application programmer in this framework provides coarse-grained interdependent tasks, and decomposes each into independent and concurrent computation modules called subtasks. These subtasks are autonomously scheduled by Unicorn runtime on the available computing devices. All network communications are layered over MPI [23].

For input and output, Unicorn tasks use an abstract entity called address space. A task may read/write any number of disjoint address spaces; each is logically shared by the subtasks but generally physically distributed across nodes in the cluster. Subtasks operate on an address space using transactional memory semantics, i.e., they check-out memory in a local view before working on it and check-in memory back to the global shared view once their computation is over. The local view visible to a subtask is user controlled and is called subtask's *memory subscription*.

For efficiency, an address space has a designated owner node that manages a distributed directory that maps addresses to locations in the cluster. As subtasks execute and write to an address, the corresponding directory entry is updated locally by the node executing the subtask. Local views are not invalidated until the end of the tasks following transactional semantics. At the end of the task, all directory changes are combined in batch mode by the address space owner. If the location of an address changes from one node to another, the former node is sent an update message. Thus, the owner node always knows the true locations of all addresses. Other nodes know true locations of addresses they have written to in a previous task. For other addresses they may not know the location and route their data transfer requests through the owner.

Unicorn uses subtask stealing to balance load. At the start of a task, Unicorn's scheduler equally divides the available subtasks among all devices in the cluster. Each device executes its assigned set and after it executes the last subtask in the set, it becomes ready to steal. It randomly selects a victim device which parts with a contiguous chunk of its outstanding subtasks and assigns them to the stealer. In case the victim has nothing to be stolen, a fail message is returned. The stealer then chooses another random victim. This continues till the task is completed.

3.2 Affinity based scheduling

In this paper, we extend Unicorn's scheduling to minimize wait for remote data by preferentially scheduling subtasks at nodes where their required data are likely to be already in the local view. To effect this we compute affinities of subtasks to nodes. Note that in Unicorn, a node's address space directory is guaranteed to contain true locations only for addresses in its local view. Hence, a node cannot compute the affinity of a subtask to other nodes. An alternative would be to force update the entire address space directory on every node (after every task) but a broadcast of this magnitude is impractical for performance reasons. Another possibility is to compute affinities of all subtasks centrally on the address space owner, whose directory entries are complete. However, a task typically uses many address spaces, each with a potentially different owner. Thus, even this alternative requires an expensive synchronization because a subtask's affinity must be based on the location of all of its data.

Hence, in this paper we explore heuristics using partial affinity information. We let each node examine the address ranges subscribed by all subtasks, only computing each subtask's affinity to itself. This is a much smaller list with size equal to the number of subtasks in one task. This can now be centrally gathered and processed. Unicorn's scheduler initially computes only the number of subtasks to assign to devices on each node, under the premise that all subtasks are created equal. We allow this step to proceed normally. Then we resort to a greedy approach to tie specific subtask IDs to devices on specific nodes. Nodes pick subtasks in a round-robin fashion. Each node picks the remaining subtask with the highest affinity score for it. If a node reaches its assigned count, it is skipped. Within the nodes, blocks of subtasks are assigned to its devices in the proportion the original scheduler determines. For our small experimental cluster of 10 nodes, the measured overhead of centrally collecting and remapping scheduler assignments is less than one millisecond. Significantly more time is spent in examining subtask subscriptions. This overhead is discussed in section 4.

Among the contributions of this paper is the computation of subtask-node affinity. A natural affinity score may be the size of data resident in the local view of the node. However,

this does not always afford the best speed-up. We also study other heuristics that consider the size of remote data instead. We study two variants of remote data affinity – one counts the number of data requests sent to remote nodes and the other that counts the number of address space pages fetched from remote nodes. Both these scores are found by first querying non-local regions from the address space directory and then combining these regions into as large contiguous chunks as possible. Unicorn's network layer allows generalized 1D and 2D data packing and we consider that in estimating the number of requests (i.e., the number of chunks) and the size of request (the total size of chunks divided by the page size).

In Unicorn, the victim assigns a number – call it s – of subtasks to the stealer on the basis of their relative rates of subtask execution (i.e., the number of subtasks executed per second before the steal operation). We retain that principle, but the actual subtasks assigned are the ones which have high affinity scores for the stealer but low scores for the victim. The victim chooses the s subtasks with the highest difference between their affinity to the stealer versus to the victim. As an aside, the stealer's affinity scores are not computed by the victim. We also do no include it with every steal request. Rather, we piggyback nodes' affinity scores on other data transfer. Since stealing happens near the end of the task, a stealer's affinity array is highly likely to reach all potential victims with negligible overhead. Nevertheless, if the affinity scores have not reached earlier, it comes with the request. In section 4, we report performance improvements with this scheme as compared to Unicorn's locality oblivious work stealer, which may allow a subtask with entire data on the victim's node to be stolen by a device on some other node with potentially no data, resulting in sub-optimal performance.

Note that evaluating all subtasks (on all nodes) for determination of affinity scores is a limitation of Unicorn's address spaces. As reported in section 4, this has non-negligible overhead. We explore evaluating fewer subtasks on all nodes. This compromises the accuracy of affinity scheduling but saves the time spent in computing affinity scores. In the next section, we analyse the impact of reducing the number of subtask subscriptions analyzed by each node.

4. Experiments and Analysis

In this section, we evaluate several node-affinity based work-stealing schedulers employing different Unicorn benchmarks. Our experiments were performed on a cluster of ten nodes, each equipped with two 6-core Intel Xeon X5650 2.67 GHz processors with 48 GB of memory. All the machines are powered with two Fermi generation Tesla M2070 GPU cards, each having 448 cores running at 1.15 GHz and 5 GB of GDDR5 memory. The machines run CentOS 6.2 with CUDA 5.5. For communication, we use Open MPI [24] 1.4.5 (over SSH) over a QDR InfiniBand [25] network.

We report our experiments on three benchmarks – image convolution, square matrix multiplication and block LU factorization. We have chosen these benchmarks as they have been well studied in the parallel domain and they make good candidates to stand for a wider range of applications. Image convolution is computationally moderate while being low on data transfer (part of which is overlapped with compute by *Unicorn's* pipeline). In contrast, matrix multiplication involves massive data transfers and is computationally expensive as well. LU factorization is an iterative interdependent series of tasks. Image convolution is also iterative: a sequence of filters is applied. The purpose of studying different kinds of applications is to understand their response to different locality-aware scheduling heuristics. All experimental results are based on three trials.

We first briefly discuss the implementation of these benchmarks over Unicorn and then study their responses to different scheduling heuristics. The results highlight that locality oblivious scheduling does not give optimal results for hybrid CPU-GPU clusters mainly because it does not account for the time spent in fetching of remote data. Our heuristics, on the other hand, attempt to minimize the time spent in data transfer.

In our image convolution experiment all color channels of a 24-bit RGB image of size $43008 * 32768$ are convolved with a $31 * 31$ filter. The input image is stored in a read-only address space (initially distributed randomly across the cluster nodes), logically divided into 336 blocks of size $2048 * 2048$. Each block is convolved using a separate subtask. However, because convolution at boundaries requires data from adjoining blocks, the input memory subscription of a subtask overlaps with other subtasks', potentially at all four boundaries. The output image is generated in a write-only address space. We convolve the filter 10 times with the image and each iteration is carried out in a different task. The experiment has a time complexity of $O(nm)$, n being the size of the image and m that of the filter.

In the matrix multiplication experiment two dense square matrices of size $2^{15} * 2^{15}$ 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. Both input matrices are initially distributed randomly across the cluster nodes. The output matrix is logically divided into $2048 * 2048$ sized 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 subtask callback is implemented using a single-precision BLAS [26] function and the GPU callback uses the corresponding CUBLAS [27] function. The experiment has 256 subtasks and each runs a computation with time complexity $O(n^3)$.

In the in-place block LU decomposition [28] experiment,

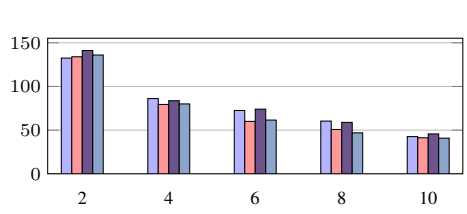
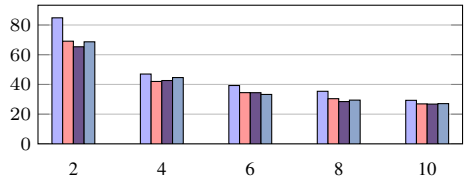


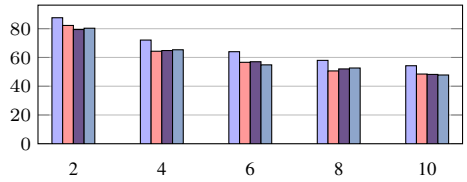
Image Convolution

Nodes	Data transferred (GB)					Data transfer events x 1000					Avg. subtask latency (secs)				
	2	4	6	8	10	2	4	6	8	10	2	4	6	8	10
Unicorn	18.2	30.5	33	36.2	35.7	6.5	14.9	18.0	20.3	21.8	6.6	3.6	2.9	2.5	2.4
Local Data	0.6	0.9	1.1	1.3	1.3	5.9	13.4	15.9	16.9	18.6	7.4	3.8	2.7	2.3	2.2
Transfer Events	9.1	21.2	33.6	37.9	38.4	2.5	7.7	12.7	16.4	19.3	7.4	3.9	2.9	2.5	2.3
Remote Data	0.6	1.0	1.2	1.2	1.3	6.5	13.1	15.0	18.0	18.9	7.5	3.8	2.7	2.3	2.2



Matrix Multiplication

Nodes	Data transferred (GB)					Data Transfer Events x 1000					Avg. subtask latency (secs)				
	2	4	6	8	10	2	4	6	8	10	2	4	6	8	10
Unicorn	8.0	23.9	36.8	49.1	55.9	0.3	1.0	1.5	2.1	2.6	7.3	7.2	7.4	7.3	6.8
Local Data	7.7	19.6	26.0	34.3	37.6	0.3	0.8	1.1	1.6	1.9	8.1	7.7	7.4	7.0	6.0
Transfer Events	7.2	21.2	28.2	34.7	38.4	0.2	0.8	1.3	1.7	2.0	8.2	8.2	7.8	6.7	6.1
Remote Data	7.3	19.9	26.5	33.8	38.9	0.3	0.8	1.3	1.6	1.9	8.2	7.8	7.7	7.0	6.5



LU Decomposition

Nodes	Data transferred (GB)					Data transfer events x 1000					Avg. subtask latency (secs)				
	2	4	6	8	10	2	4	6	8	10	2	4	6	8	10
Unicorn	15.2	28.6	35.6	41.1	44.7	1.0	1.8	2.3	2.6	2.8	4.3	3.9	3.7	3.5	3.6
Local Data	6.3	16.4	22.2	25.6	28.2	0.4	1.1	1.4	1.6	1.8	4.6	4.4	4.1	3.9	3.7
Transfer Events	5.9	16.1	22.0	25.8	28.8	0.4	1.0	1.4	1.7	1.9	4.6	4.3	3.9	3.8	3.8
Remote Data	6.3	16.1	21.1	25.9	27.8	0.4	1.0	1.4	1.6	1.8	4.5	4.3	4.0	3.9	3.8

Fig. 1: Locality aware scheduling - Graphs plot Execution Time (secs) versus Nodes

the input matrix ($2^{15} * 2^{15}$) is kept in a read-write address space (initially distributed randomly across the cluster nodes) and is logically divided into $2048 * 2048$ sized blocks. The matrix is solved top-down for each of the 16 diagonal blocks. 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. Time complexities of these tasks are $O(n)$, $O(n^2)$ and $O(n^3)$, respectively. 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 within 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.

All our experiments are written with no particular spatial ordering of subtasks. Thus, the adjacent subtasks of these experiments do not necessarily execute on adjacent address space regions. This makes a better case for studying the effectiveness of affinity heuristics. The Image convolution experiment has the smallest memory footprint with a total input size of 3.94 GB, followed by LU decomposition with 4 GB input. Matrix Multiplication has two input address spaces of 4 GB each making the total input size 8 GB.

For the three benchmarks, Figure 1 plots the performance of *Unicorn's* locality-oblivious scheduler as well as the performance of the *local data* based affinity and compares these to *remote data* based affinity (*transfer events* and *remote data*). The figure also records the cluster-wide data transfers and subtask latency incurred in these experiments.

Results show that all our heuristics perform better than *Unicorn's* locality oblivious scheduler at nearly all data points. For image convolution experiment, a maximum gain of 22.3% (over *Unicorn's* scheduler) is observed with *remote data* heuristic for the eight node case. This is attributed to a massive data transfer reduction from 36.2 GB (for Unicorn) to 1.18 GB. A gain of similar magnitude is not reflected in execution time because much of the transfer latency is hidden behind other computation for the experiment. We find that Unicorn's work stealing results in most of the subtasks actually being executed by GPUs (being more powerful than CPUs for a SIMD computation like image convolution) where computation of a subtask is overlapped with the data transfer of the next. Even reducing the overall data transfer by more than 96% does not make this compute bound pipeline of GPU subtasks any faster.

For the matrix multiplication experiment, we observe the maximum gain of 19.56% for the eight node case with the *transfer events* heuristic. This result is attributed to a 29.25% reduction in data transfer, a 22% reduction in data transfer events and a 0.6 sec gain in average subtask latency. Note that this is a communication bound experiment and the percentage reduction in data transfer has resulted in a similar gain in performance after accounting for the 9.6% overhead

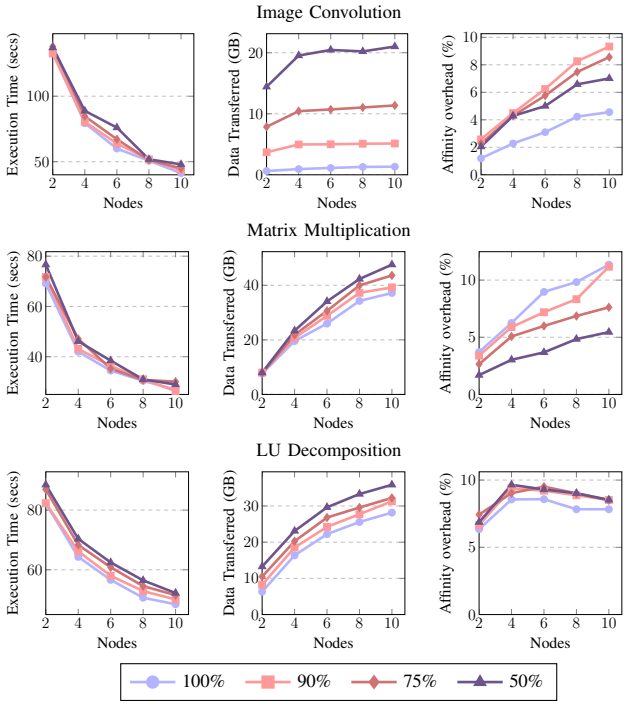


Fig. 2: Incremental affinity subtask reduction

in affinity computation.

For the LU decomposition experiment, the maximum gain of 12.67% is achieved by the *local data* heuristic for the eight node case. This heuristic resulted in a 37.6% reduction in total data transfer and has a reported overhead of 8%. The experiment has moderate performance gains as compared to data transfer savings as it is an iterative experiment, with a mix of compute and communication bound tasks per iteration.

All our heuristics perform fairly closely to each other. *Local data* heuristic reports the lowest execution time for experiments with small memory footprint (image convolution and LU decomposition) when the number of nodes involved is not more than six. When the number of nodes increases to eight or ten, the *remote data* heuristic outperforms others. As far as reduction in data transfer is concerned (in comparison to Unicorn), we expect iterative experiments like image convolution and LU decomposition to do better than the non-iterative matrix multiplication like experiments as the benefits of reducing data transfers are realized every iteration. We find that *transfer events* heuristic is not able to bring down data transfers in image convolution by the same margin as other heuristics, resulting in its poor performance as compared to the other two. This shows that an indirect heuristic that optimizes data transfer events in an attempt to reduce actual data transfers may not be as suitable as compared to other heuristics that directly target maximizing local or minimizing remote data.

Despite the performance improvements with our heuristics, we observe significant overhead in affinity determination

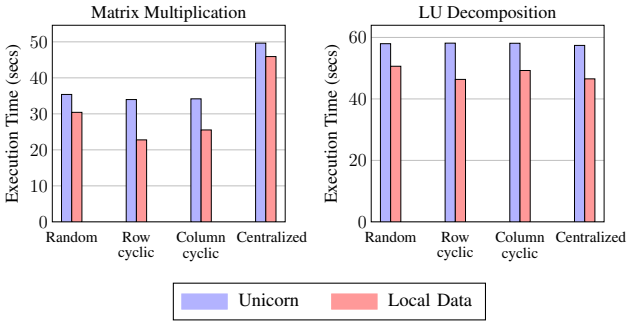


Fig. 3: Impact of initial data distribution pattern

(Figure 2). This is because our affinity determination algorithm evaluates input memory subscriptions of all subtasks of the application task on all nodes. In order to reduce this overhead, we reduce the number of subtasks evaluated per node (for affinity determination) and study the response of *local data* heuristic to this change (Figure 2). The figure plots execution time, data transferred and affinity determination overhead when the number of subtasks evaluated (for affinity computation) per node are gradually reduced from 100% to 50%. Results show that for all experiments, reduction in the number of evaluated subtasks also reduces the effectiveness of the heuristics. On an average, image convolution becomes 11.85% slower when the number of analyzed subtasks at each node reduces from 100% to 50%. Similarly, the impact on matrix multiplication is 8.33% and on LU decomposition is 9.31%. In general, the magnitude of loss in performance despite gains in the affinity task's overhead makes this overall less profitable.

As stated earlier, all our tasks have a random block distribution of the initial data in their address spaces. A good affinity scheduler should be agnostic to the changes in initial data availability pattern. We study *local data* heuristic with different initial placements of data and compare its performance to that of Unicorn's scheduler. Figure 3 evaluates matrix multiplication and LU decomposition (for eight node case) for various schemes like *centralized* (entire address space on one cluster node), *row cyclic* (rows of $2048 * 2048$ blocks placed in sequence on all cluster nodes), *column cyclic* (columns of $2048 * 2048$ blocks placed in sequence on all cluster nodes) and the default *block random* ($2048 * 2048$ blocks placed randomly on any cluster node). Results show that our runtime maintains performance despite the changes in data availability pattern. At all data points, our heuristic results in better performance than Unicorn's scheduler. Further, for the matrix multiplication experiment more favorable distributions like *row cyclic* and *column cyclic* achieve proportionately higher gains with our heuristic in comparison to other less favorable distributions.

Lastly, we study the effectiveness of our steal policy post the initial distribution of subtasks. In the absence of external factors, a good initial subtask distribution is effective in balancing load for a large fraction of the task. However,

towards the end of a task a few devices finish early and try to steal work from others. In this case, we let the victim assign those subtasks to the stealer that are high on affinity scores for the stealer but low on affinity scores for the victim. As compared to Unicorn's affinity oblivious steal, this scheme reduces data transfers by around 2.5% on an average for all three experiments when affinity scores are computed with *remote data* heuristic. This translates into performance improvements up to 3.0%. When affinity scores are computed with *local data* heuristic, however, we observe mostly flat response with our scheme. For image convolution experiment, it reports around 2.5% performance gain as compared to Unicorn's stealing while a 4.3% degradation is observed with matrix multiplication.

5. Conclusions and Future Work

We present a study of locality aware work stealing in the context of distributed shared memory programming on hybrid CPU-GPU clusters. In particular, we augment *Unicorn's* work-stealing scheduler with data locality and study its characteristics. We evaluate two other heuristics that attempt to reduce the time spent in transfer of non-local data across the cluster. These heuristics, respectively minimize the number of data transfer events and the size of remote memory fetched. These heuristics are based on each node computing affinity based only on the data it has. The results demonstrate reasonable performance improvements with these heuristics despite non-trivial overhead in address subscription analysis. Given that Unicorn hides some network latency behind other computation, sometime the gain in overall application speed does not reflect the significant data transfer reduction, but in a loaded network this gain can be useful. We believe that with better analysis of subscription, it is possible to improve affinity scores further.

All the benchmarks presented in this paper are regular. However, we have experimented with a few irregular applications like page rank and initial results are promising, even if incomplete at this time. We believe that experimenting with more irregular applications and using larger clusters may promote better understanding of these heuristics.

References

[1] Beri, Bansal, and Kumar, "A scheduling and runtime framework for a cluster of heterogeneous machines with multiple accelerators," in *Proceedings of the 2015 IEEE 29th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '15, 2015.

[2] Nieplocha *et al.*, "Advances, applications and perf. of the global arrays shared memory programming toolkit," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, May 2006.

[3] Charles *et al.*, "X10: An object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*.

[4] Acar, Blelloch, and Blumofe, "The data locality of work stealing," in *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '00.

[5] Philbin *et al.*, "Thread scheduling for cache locality," in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS VII, 1996, pp. 60–71.

[6] Tam, Azimi, and Stumm, "Thread clustering: Sharing-aware scheduling on smp-cmp-smt multiprocessors," in *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07, 2007, pp. 47–58.

[7] McGregor, Antonopoulos, and Nikolopoulos, "Scheduling algorithms for effective thread pairing on hybrid multiprocessors," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS '05.

[8] Vaswani and Zahorjan, "The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors," in *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '91.

[9] "Intel Threading Building Blocks," http://www.threadingbuildingblocks.org/.

[10] Guo *et al.*, "Slaw: A scalable localityaware adaptive work-stealing scheduler," in *In 24th IEEE International Symposium on Parallel and Distributed Processing*, ser. IPDPS '10.

[11] Huang *et al.*, "Enabling locality-aware computations in openmp," *Sci. Program.*, vol. 18, no. 3-4, Aug. 2010.

[12] Dagum and Menon, "OpenMP: An industry-standard API for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Jan. 1998.

[13] Nugteren, Braak, and Corporaal, "A study of the potential of locality-aware thread scheduling for gpus," in *Euro-Par 2014: Parallel Processing Workshops*, ser. Lecture Notes in Computer Science, 2014, vol. 8806.

[14] SUGIMOTO, INOb, and HAGIHARA, "Improving cache locality for gpu-based volume rendering."

[15] Unkule, Shaltz, and Qasem, "Automatic restructuring of gpu kernels for exploiting inter-thread data locality," in *Proceedings of the 21st International Conference on Compiler Construction*, ser. CC'12, 2012.

[16] Lee *et al.*, "Locality-aware mapping of nested parallel patterns on gpus," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, Dec 2014.

[17] Quintana-Ortí *et al.*, "Solving dense linear systems on platforms with multiple hardware accelerators," *SIGPLAN Not.*, vol. 44.

[18] "Magma 1.4.1," http://icl.cs.utk.edu/magma/, 2013.

[19] 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.

[20] Gautier *et al.*, "Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures," in *IPDPS '13*, ser. IPDPS '13, 2013, pp. 1299–1308.

[21] Yin *et al.*, "Slam: Scalable locality-aware middleware for i/o in scientific analysis and visualization," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC '14, 2014.

[22] Ng *et al.*, "Visdsi: Locality aware i/o solution for large scale data visualization," in *Utility and Cloud Computing (UCC), 2013 IEEE/ACM 6th International Conference on*, Dec 2013.

[23] Gropp *et al.*, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Comput.*, vol. 22, no. 6, pp. 789–828, Sep. 1996.

[24] Gabriel *et al.*, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Euro. PVM/MPI Users Group Meeting*, 2004, pp. 97–104.

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

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

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

[28] Demmel, Higham, and Schreiber, "Block LU factorization," http://www.netlib.org/utk/papers/factor/node7.html, 1995.