Distributed Perception by Collaborative Robots

Ramyad Hadidi[1], Jiashen Cao[1], Matthew Woodward[1], Michael S. Ryoo[2], and Hyesoon Kim[1]

***Abstract*—Recognition ability and, more broadly, machine learning techniques enable robots to perform complex tasks and allow them to function in diverse situations. In fact, robots can easily access an abundance of sensor data that is recorded in real time such as speech, image, and video. Since such data is time sensitive, processing it in real time is a necessity. Moreover, machine learning techniques are known to be computationally intensive and resource hungry. As a result, an individual resource-constrained robot, in terms of computation power and energy supply, is often unable to handle such heavy real-time computations alone. To overcome this obstacle, we propose a framework to harvest the aggregated computational power of several low-power robots for enabling efficient, dynamic, and real-time recognition. Our method adapts to the availability of computing devices at runtime and adjusts to the inherit dynamics of the network. Our framework can be applied to any distributed robot system. To demonstrate, with several Raspberry-Pi3-based robots (up to 12) each equipped with a camera, we implement a state-of-the-art action recognition model for videos and two recognition models for images. Our approach allows a group of multiple low-power robots to obtain a similar performance (in terms of the number of images or video frames processed per second) compared to a high-end embedded platform, Nvidia Tegra TX2.**

***Index Terms*—Deep Learning in Robotics and Automation, Distributed Robot System**

I. INTRODUCTION

THE availability of larger datasets, improved algorithms, and increased computing power is rapidly advancing the applications of deep neural networks (DNNs). This advancement has extended the capabilities of machine learning to areas such as computer vision [1], natural language processing [2], neural machine translation [3], and video recognition [4], [5]. In the meantime, robots have access to an abundance of data from their environment and are in desperate need to extract useful information for enhanced handling of complex situations. While robots can benefit tremendously from DNNs, satisfying their intensive computation and data requirements is a challenge for robots. These challenges are even exacerbated in resource-constrained devices, such as low-power robots, mobiles, and Internet of things (IoT) devices, and a significant amount of research efforts has been invested to overcome them [6]–[10], such as collaborative computation between edge devices and the cloud [11]–[13], or customized mobile implementations [14]–[20]. Despite all these efforts, scaling current DNNs to robots and processing generated data in *real time* faces challenges due to limited computing power and energy supplies in robots. Hence, in order to handle current and future DNN applications that are more resource hungry [21]–[23] and extract useful information from raw data in a timely manner, creating an efficient solution is critical.

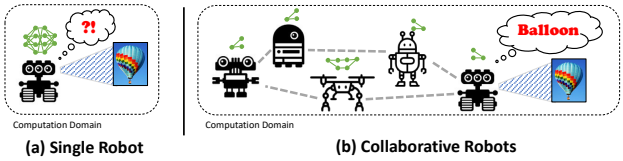


Fig. 1: Collaborative robots performing distributed inference.

Our main idea is to utilize the aggregated computational power of robots in a distributed robot system to perform DNN-based recognition in real time. Such collaboration enables robots to take advantage of the collective computing power of the group in an environment to understand the collected raw data, while none of the robots would experience energy shortage. Although such collaboration could be extended to a variety of systems, limited computing power and memory, scarce energy resources, and tight real-time performance requirements make this challenge unique to robots. In this paper, we propose a technique for collaborative robots to perform cost-efficient, real-time, and dynamic DNN-based computation to process raw data (Figure 1). Our proposed technique examines and distributes a DNN model to gain high real-time performance, the number of inferences per second. We explore both *data parallelism* and *model parallelism*, where data parallelism consists of processing independent data concurrently and model parallelism consists of splitting the computation across multiple robots. For demonstration, we use up to 12 GoPiGos [24], which are Raspberry-Pi3-based [25] robots, each with a camera [26] (Figure 2). As an example DNN, to detect an object and related types of actions happening in an environment, we implement a state-of-the-art action recognition model [4] with 15 layers and two popular image recognition models, AlexNet [1] and VGG16 [22].

The summary of our contributions in this paper is as follows: (i) We develop a profiling-based technique to effectively dis-

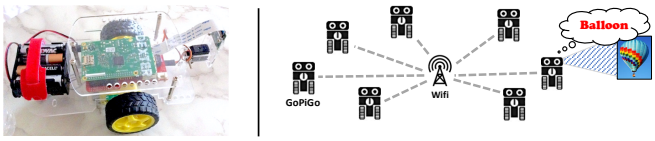


Fig. 2: Our GoPiGo distributed robot system.

Manuscript received: February, 24, 2018; Revised May, 23, 2018; Accepted June, 19, 2018.

This paper was recommended for publication by Editor Nak Young Chong upon evaluation of the Associate Editor and Reviewers' comments. *This work was supported by Intel.

[1]Ramyad Hadidi, Jiashen Cao, Matthew Woodward, and Hyesoon Kim are with Computer Science School and Electrical Engineering Department, Georgia Institute of Technology, GA 30332, USA {rhadidi,jcao62,mwoodward}@gatech.edu, hyesoon@cc.gatech.edu.

[2]Michael S. Ryoo is with EgoVid Inc., Ulsan 44919, Korea mryoo@egovid.com

Digital Object Identifier (DOI): see top of this page.

TABLE I: Comparison with recent related work.

	End-Compute Device	Number of Devices	Localized Inference	Real-Time Data Process	Partitioning Mechanism	Model- & Data-Parallelism	Runtime Adaptability
Neurosurgeon [11]	Tegra TK1 [27]	1	✗	✗	Inter-Layer	✗	✗
MoDNN [28]	LG Nexus 5	4	✓	✗	Intra-Layer	✗	✗
DDNN [13]	✗	Many	✗	✓	Inter-Layer	Data Parallelism	✗
Our Method	Raspberry Pi [25]	Many	✓	✓	Intra- & Inter-Layer	Both	✓

tribute DNN-based applications on a distributed robot system while considering memory usage, communication overhead, and real-time data processing performance. (ii) We propose a technique that dynamically adapts to the number of available collaborative robots and is able to interchange between the robot, which inputs data, and computational robots. (iii) We apply our technique on a distributed robot system with Raspberry-Pi3-based hardware, investigating a state-of-the-art action and two image recognition DNN models.

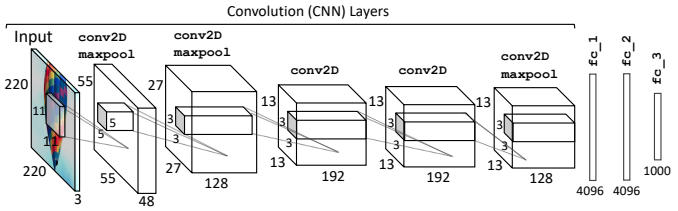
II. Related Work

Performing distributed perception with collaborative robots is a new concept; however, various related research to process DNN applications for real-time performance has been done. One of the first papers to distribute computation is [29]; however, it investigates such distribution and partitioning specific for training and not inference while only focusing on high-performance hardware. A recent work, Neurosurgeon [11], dynamically partitions a DNN model between a *single* edge device (Tegra TK1, $200) and the cloud for higher performance and better energy consumption. Neurosurgeon does not study the collaboration between edge devices and is dependent on the existence of a cloud service. A similar study of partitioning the computations between mobile and cloud is also done in [12] using the Galaxy S3. Another work, MoDNN [28], creates a local distributed mobile computing system and accelerates DNN computations. MoDNN uses only mobile platforms (LG Nexus 5, $350) and partitions a DNN using input partitioning within each layer, especially by relying on sparsity in the matrix multiplications. However, MoDNN does not consider real-time performance because its most optimized system with four Nexus-5 devices has a latency of six seconds. DDNN [13] also aims to distribute the computation in local devices. However, in its mechanism, in addition to retraining the model, each sensor device performs the first few layers in the network and the rest of the computation is offloaded to the cloud system. Therefore, similar to [11], [12] is dependent on the cloud. Table I provides a comparison of these works with our method. Additionally, executing DNN models in resource-constrained platforms has recently gained great attention from industry, such as ELL library [14] by Microsoft and Tensorflow Lite [19] by Google. However, these frameworks are still in development and have limited capability. Our work is different because (i) we study cost-efficient distributed robot systems, (ii) we examine conditions and methods for real-time processing of DNNs, and (iii) we design a collaborative system with many devices.

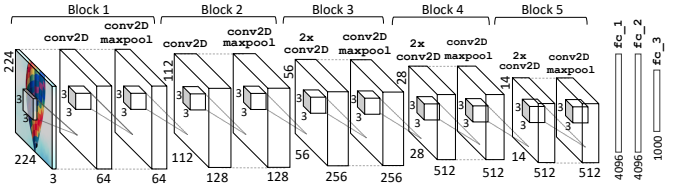
III. BACKGROUND

In the past three years, the use of DNN for robots has become increasingly popular. This not only includes robot perception of objects [30], [31] and actions [32], but also robot action policy learning [33], [34] using DNNs. This section gives an overview of common DNN layers and models we use for object and action recognitions. DNN models are composed of several layers stacked together for processing inputs. Usually, first layers are convolution layers `(conv)`, which consist of a set of filters that are applied to a subset of inputs by sweeping each filter (i.e., kernel) over them. To introduce non-linearity, activation layers `(act)` apply a non-linear function, such as ReLU, $f(x) = max(0,x)$, allowing a model to learn complex functions. Sometimes, a pooling layer, such as a max pooling layer `(maxpool)`, downsamples the output of a prior layer and reduces the dimensions of data. Finally, a few fully connected (dense) layers `(fc)` perform a linear operation of weighted summation. A fully connected layer of size n has n set of weights and creates an output of size n. Among the mentioned layers, `fc` and `conv` layers are among the most compute- and data-intensive layers [35]. Hence, our technique aims at alleviating the compute cost and overcoming the resource barriers of these layers by distributing their computation.

Image-based Object Recognition Models: Recent advancements in computer vision [36] have allowed us to achieve high accuracies and surpass human-level accuracy [37]. Computer vision models extensively use `conv` layers, the heavy computations of which are not ideal for low-power robots [38]. For demonstration, we studied AlexNet [1] and VGG16 [22], the models of which are shown in Figure 3.



(a) Single stream AlexNet model.



(b) VGG16 model.

Fig. 3: Image recognition models.

Action Recognition Model: Recognizing human activities and classifying them (i.e., action recognition) in videos is a challenging task for DNN models. Such DNN models, while performing still image classification, must also consider the temporal content in videos. We use the model of Ryoo et al. [4], which consists of two separate recognition streams,

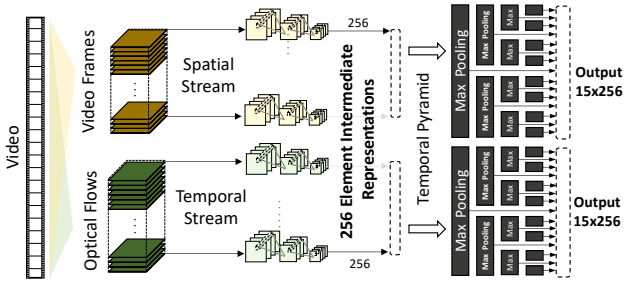


Fig. 4: Temporal pyramid generation.

spatial and temporal convolution neural networks (CNNs), the outputs of which are combined in a temporal pyramid [39] and then fused in fully connected layers to produce predictions.

(a) Spatial Stream CNN: The spatial stream, similar to image recognition models that classify raw still frames from the video (i.e., images), is implemented with `conv` layers. This model, as input, takes a frame of size 16×12×3 (in RGB) and processes it with three `conv` layers, each with 256 filters, the kernel sizes of which are 5×5, 3×3, and 3×3, respectively. Then, features of each frame are summarized in a 256-element vector. Since this stream processes still images, for training, we can use any representative dataset, such as ImageNet [36], by adding a dummy output dense layer.

(b) Temporal Stream CNN: The temporal stream takes optical flow as input, which explicitly describes the motion between video frames (we use Färenback [40] algorithm). In other words, for every pixel at a position (u_t, v_t) at time t, the algorithm finds a displacement vector $\mathbf{d}_t$ for each pair of consecutive frames, or $\mathbf{d}_t = (d_t^x, d_t^y) = (u_{t+\Delta t} - u_t, v_{t+\Delta t} - v_t)$. We compute the optical flow for 10 consecutive frames and stack their (d_t^x, d_t^y) to create an input with the size of 16×12×20. Subsequently, the data is processed with three `conv` layers, each with 256 filters, the kernel sizes of which are 5×5, 3×3, and 3×3, respectively. Finally, the features are summarized in a 256-element vector. By adding a dummy output dense layer, we can train the temporal stream with any video dataset, such as HMDB [41].

(b) Temporal Pyramid: To generate a single representation from the two streams, a single spatio-temporal pyramid [39] is generated for each video. Figure 4 depicts the steps of generating a four-level temporal pyramid from a video. Such a pyramid structure of `maxpool` layers creates an output with a fixed size that is agnostic to the duration of videos. For each stream, 15 `maxpool` layers with different input ranges generate a 15×256 output. Finally, the data with size 2×15×256 is processed by two `fc` layers with sizes of 8192, and an `fc` layer with the size of 51 outputs HMDB classes.

IV. DISTRIBUTING DNN

In this section, we examine our distribution and parallelization methods for computation of a DNN model over multiple low-power robots (i.e., devices). We examine this problem in the context of real-time data processing, which means a continuous stream of raw data is available. Our goal is to reduce the effective process time per input data. As terminology, a *task* is the processes that are performed on input data by a layer or a group of consecutive layers. We

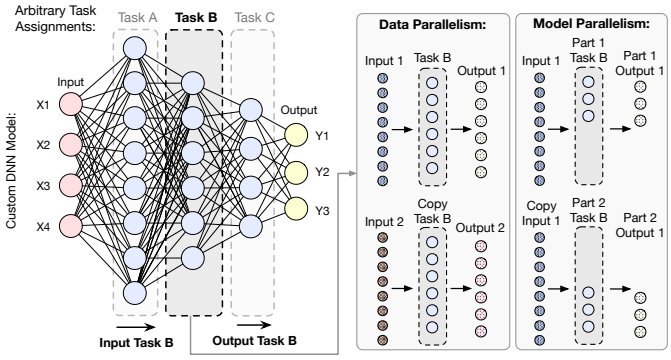


Fig. 5: Model and data parallelism for task B on two devices.

introduce *data parallelism* and *model parallelism* (inspired by concepts in GPU training [42]), which are applicable to a task. Data parallelism is duplicating devices that perform the same task, or share the same model parameters. Model parallelism is distributing a task, which is dividing the task into sub parts and assigning them to additional devices. Thus, in model parallelism, since the parameters of the model are divided among devices, the parameters are not shared.

Figure 5 depicts model and data parallelism of task B, an arbitrary task, for two devices in an example DNN with three layers. Data parallelism basically performs the same task on two independent inputs, while in model parallelism, one input is fed to two devices that perform half of the computations. To create the output, a merge operation is required (for now, we assume inputs are independent, see §V-C). Implementing data parallelism starts with assigning each newly arrived data to devices. However, performing model parallelism requires a knowledge of deep learning. In fact, the effectiveness of model parallelism depends on factors such as the type of a layer, input and output sizes, and the amount of data transfer. Furthermore, the performance is tightly coupled with the computation balance among devices, whereas, in data parallelism, the computations are inherently balanced. We investigate these methods for `fc` and `conv` layers since these layers demand the most computations and resources.

Fully Connected Layer: In an `fc` layer, the value of each output is dependent on the weighted sum of all inputs. To apply model parallelism to this layer, we can distribute the computation of each output while transmitting all the input data to all devices. Since the model remains the same, such a distribution does not require training new weights. Later, when each subcomputation is done, we need to merge the results for the consumption of the next layer. As an example of how model and data parallelism affect the performance, we examine various `fc` layers, the input sizes of which are 7,680, but with different output sizes. For each layer, we

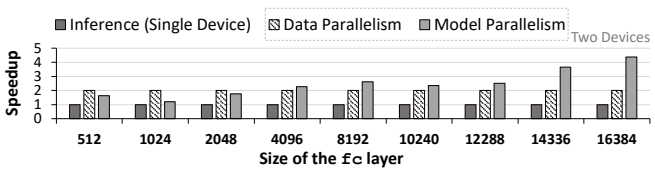


Fig. 6: Performance (i.e., throughput) speedup of model and data parallelism on two Raspberry Pis executing an `fc` layer.

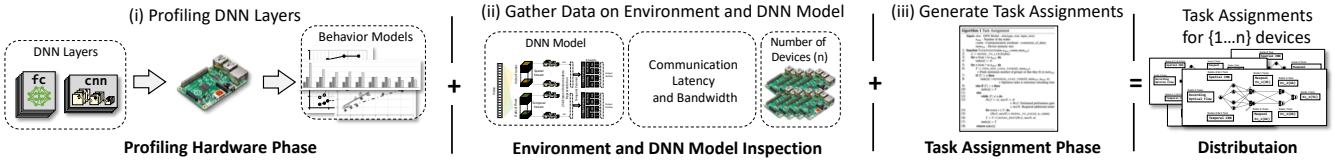


Fig. 7: Steps for generating task assignments in our solution.

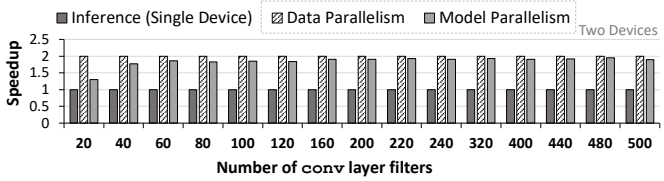


Fig. 8: Performance speedup of model and data parallelism on two Raspberry Pis executing a `conv` layer.

measure its performance (i.e., throughput) on a Raspberry Pi 3 (Table II). Figure 6 illustrates the performance of model and data parallelism normalized to performing the inference on a single device. As we see, for `fc` layers larger than 10,240, model parallelism performs better. In fact, after examining the performance counters of processors, we find that processors start using the swap space for `fc` layers larger than 10,240. Since in model parallelism a layer is distributed on more than one device, we reduce memory footprint and avoid swap space activities, which results in speedups greater than 2×.

Convolution Layer: Since computations between the filters of a `conv` layer are independent, distributing the computations has various forms, such as distributing filters while copying the input, dividing input while copying all filters, or a mix of these two. In fact, such methods of distributions are already integrated in many machine learning frameworks to increase data reuse and therefore decrease execution time. To gain insights, we examine a series of `conv` layers with the input size of 200×200×3 and the kernel size of 5×5, with different numbers of filters in Figure 8. As seen, the performance of data parallelism is always better than that of model parallelism, because while model parallelism pays the high costs of merging and transmitting the same inputs, for data parallelism, frameworks optimize accesses better for high data reusability.

V. PROPOSED SOLUTION

A. Task Assignments

To find a close to optimal distribution for each DNN model, given the number of devices in the system, we devise a solution based on profiling. Our goal is to increase the number of performed inferences per second, or *IPS*. As discussed in §IV, profiling is necessary for understanding the performance benefits of data and model parallelism. In other words, we must consider whether assigning more than one task to any device will cause significant slowdown because of the limited memory resource or if data or model parallelism with its overheads, such as data transmits and merges, increases IPS. In our solution, Figure 7, first, for each layer, we profile execution times and memory usages of its original, model-parallelism, and data-parallelism *variants.* For each hardware system, the profiling is performed offline and only once for creating this data. Second, our solution takes the target DNN model, number of devices, and communication overhead (a regression model of latency based on the data size). Finally, using gathered data, we generate task assignments based on the flow of Algorithm 1.

Algorithm 1 Task Assignment Algorithm.

1: **function** TASKASSIGNMENT($dnn, n_{max}, comm, mem_{size}$)
   Inputs: dnn: DNN model in form of layers[type, size, $input_{size}$, β]
   n_{max}: Maximum number of the devices
   $comm$: Communication overhead model ($comm(size_{data})$)
   mem_{size}: Device memory size
2: $L :=$ EXTRACT_MODEL_TO_LAYERS(dnn)
3: **for** n from 1 to n_{max}: **do**
4: $tasks_{final}[n] := \varnothing$
5: **for** n from 1 to n_{max}: **do**
6: TG, $noFit :=$ FIND_INITIAL_TASKGROUP(L, mem_{size})
7: **if** $sizeof(TG) > n$ **then**
8: tasks[n] =COMBINE_TASKS(TG, mem_{size}, n_{max}, n)
9: **if** $sizeof(TG) = n$ **then**
10: tasks[n] = TG
11: **if** $sizeof(TG) < n$ or $noFit == True$ **then**
12: **while** $sizeof(TG) \neq n$ **do**
13: $task_{variant} := \varnothing$
14: **for** every $t \in TG$: **do**
15: [$task_{variant}$] += PROFILED_VARIANTS(t, $comm$)
16: $task_{replaced}$, $task_{new}$ = SELECT_LOWEST([$task_{variant}$])
17: $TG = TG - task_{replaced} + task_{new}$
18: $tasks_{final}[n] = TG$
19: **return** $tasks_{final}$

In this algorithm, the function in line 2 extracts the model input, *dnn*, into layers, L, while also accounting for buffering requirements (i.e., sliding windows>1, see §V-C). Required extra buffers should be specified by the user in β. Because of the possibility that during execution some devices are inactive, busy, or have more than one input, we generate task assignments offline for all the possible number of devices (e.g., one, two, ..., total number of devices). For every number of devices, n, we create a dictionary of the node's name to its tasks, $tasks_{final}[n]$, and initialize it in Line 4 to the empty set. Then, from Line 5, we start a for loop for generating task assignments for the n number of devices. Since we generate all of the task assignments for any number of devices offline, our system can dynamically change the number of devices without the cost of computing a new assignment. To do so, first, the function in Line 6 generates an initial tasks group, TG, from L, such that every entity in TG fits in mem_{size} of our devices. Basically, the function starts from the first layer while using the profiled data and creates a group of consecutive layers until they cannot fit in the mem_{size}, and then moves on for creating the next group. (If a single layer does not fit in the memory, *noFit* flag is set for that entity in TG.) Then, based on the number of initial tasks groups, $sizeof(TG)$, the algorithm changes TG by adding or removing tasks until all n nodes are utilized, or $sizeof(TG) = n$. If

$sizeof(TG) > n$, it means current tasks need more devices than what the system has, so we have to co-locate some tasks together and pay the overhead of task reloads. Hence, the function in Line 8 tries to combine two consecutive tasks (two tasks such that one produces data and the other consumes it directly) that together have the lowest memory consumption across all possible consecutive tasks and performs the process until the tasks fit on n devices. This is because lowest memory consumption is directly related to the lower reloading time of tasks to the memory. If $sizeof(TG) < n$ (or *noFit* is set), the function in Line 15 uses the profiled data and the communication model, *comm*, to estimate the execution time of new task variants, $task_{variant}$, for all variants of the task, that is, original, model- and data-parallelism variants. Then, Line 16 chooses the variant with the lowest execution time across all possible variants for all tasks and outputs the to-be-replaced task ($task_{replaced}$) and the selected variant ($task_{new}$). Finally, Line 17 updates TG. This process continues in the while loop (Line 12) until we utilize all available devices, or $sizeof(TG) = n$. In this algorithm, since performance gain and communication overhead are estimations, optimality is not guaranteed. However, since task assignment is not in the critical path, we can fine-tune assignments before deployment (fine-tuning is not performed in our experiments).

B. Dynamic Communication

In our solution, devices need to communicate with each other efficiently for transmitting data and commands. We use Apache Avro [43], a remote procedure call (RPC) and data serialization framework in our solution. The RPC capacity of Avro enables us to request a service from a program located in another device. In addition, Avro's data serialization capability provides flexible data structures for transmitting and saving data during processing while preserving its format. Therefore, a device may offload the results of a computation to another device and initiate a new process. To effectively identify all devices, each device has a local copy of a shared IP address table from which its previous and next set of devices and its assigned task are identified. Furthermore, to adapt to the dynamics of the environment, a master device may update the IP table based on the generated task assignments. Similar to any network, we allocated a buffer of incoming data on all the devices. Whenever a buffer is almost full, the associated device (i) sends a signal to the previous devices, which permits them to drop some unnecessary input data (i.e., reducing sampling frequency), and (ii) sends a notification the master device. Afterward, the master device, based on such notification and the availability of devices, may update the IP table to achieve better performance (in our experiments, updates stop real-time processing for $\leq$minute).

C. Sliding Window

Our action recognition model processes a whole video for each inference. However, in reality, the frames of a video are generated by a camera (30 FPS). To adapt a model for real-time processing, we propose the use of a sliding window over the input and intermediate data, whenever needed, while

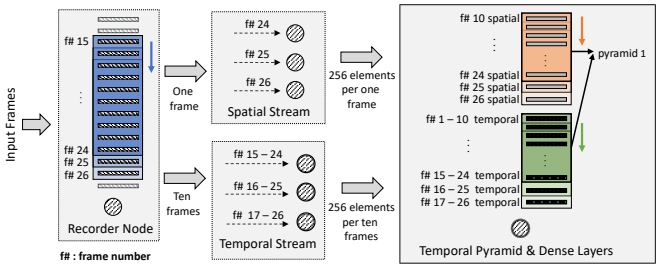


Fig. 9: Sliding window for an example system of eight devices. While some tasks require sliding window, with different sizes, others may not need it.

TABLE II: Raspberry Pi 3 specifications [25].

CPU	1.2 GHz Quad Core ARM Cortex-A53	
Memory	900 MHz 1 GB RAM LPDDR2	
GPU	No GPGPU Capability	
Price	\$35 (Board) + \$5 (SD Card)	
Power Consumption	Idle (No Power Gating)	1.3 W
	%100 Utilization	6.5 W
	Averaged Observed	3 W

distributing the model. For instance, the temporal stream accepts an input of optical flows from 10 consecutive frames, so a sliding window of size 10 over the recent inputs is required. In a sliding window, whenever new data arrives, we remove the oldest data and add the new data to the sliding window. Note that to order arriving data, a unique tag is assigned to each raw data during recording time. Figure 9 illustrates this point with an example of eight devices in a system. The recorder device keeps a sliding window of size 10 to supply the data, while the devices that process spatial and temporal streams do not have a sliding window buffer. On the other hand, since the temporal pyramid calculation requires a spatial data of 15 frames and temporal data of 25 frames, the last device keeps two sliding window buffers with different sizes. We can extend the sliding window concept to other models that have a dependency between their inputs to create a continuous data flow. Furthermore, the sliding window is required to enable data and model parallelism. This is because a device needs to order its input data while buffering arrived unordered data.

VI. EVALUATION

We evaluate our method on distributed Raspberry-Pi-based [25] (Table II) robot (GoPiGo [24]). Furthermore, we compare our results with two localized implementations: (i) a high-performance (HPC) machine (Table III) and (ii) Jetson TX2 [44] (Table IV). For all implementations, we use Keras 2.1 [45] with the TensorFlow GPU 1.5 [46]. We measure power consumption of all modules, except mechanical parts, with a power analyzer. A local WIFI network with the

TABLE III: HPC machine specifications.

CPU	2x 2.00GHz 6-core Intel E5-2620	
Memory	1333 MHz 96 GB RAM DDR3	
GPU	Titan Xp (Pascal) 12 GB GDDR5X	
Total Price	\$3500	
Power Consumption	Idle	125 W
	%100 Only-CPU Utilization	240 W
	%100 Only-GPU Utilization	250 W

TABLE IV: Nvidia Jetson TX2 specifications [44].

CPU	2.00GHz Quad Core ARM Cortex-A57 2.00GHz Dual Denver 2	
Memory GPU Total Price	1600 MHz 8 GB RAM LPDDR4 Pascal Architecture - 256 CUDA Core $600	
Power Consumption	Idle (Power Gated) %100 Utilization Averaged Observed	5 W 15 W 9.5 W

measured bandwidth of 62.24 Mbps and a measured client-to-client latency of 8.83 ms for 64 B is used. We use a measured communication model of $t = 0.0002d + 0.002$, in which t is latency (seconds) and d is the data size (kB). All trained weights are loaded to each robot's storage, so each robot can be assigned to any task.

A. Single Robot

Since a single robot has limited memory, it usually cannot handle the execution of all the tasks efficiently because for performing any computation, data should be loaded to memory from storage. Figures 10a and b show the loading time and memory usage of general tasks in the action recognition model. The memory requirement of dense layers is larger than 1 GB, so a single robot needs to store and load intermediate states (i.e., activations of a layer) to its storage, which incurs high delays. To gain insight, we even try a dense layer with half-sized dimensions of the original one, with 15% lower accuracy. Figure 10 shows that, in this case, even with a negligible computation time, the overhead of loading each task is high for real-time processing. Even when assuming zero loading time, as in Figure 10c and d depict for energy and inference time, the inference time of the half-sized `fc` layer is more than 0.7 seconds, while its energy per inference is 10× larger than that of spatial/temporal streams. Hence, in such an implementation, we still cannot process data in real time.

B. Action Recognition

In the action recognition model, the recording robot also computes optical flow, the computation of which is not heavy (e.g., 4 ms for 100 frames using the method in [40]). Each robot manages a sliding window buffer, explained in §V-C, the

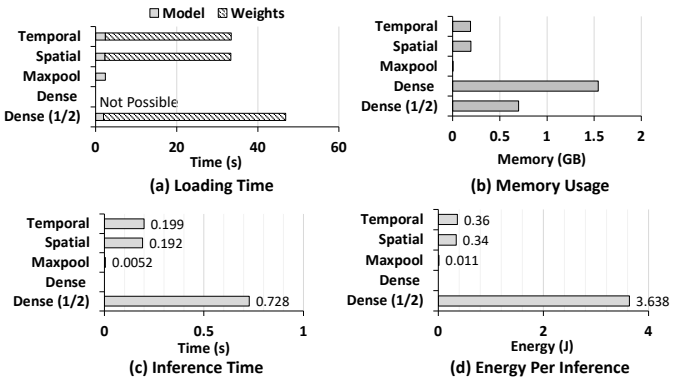
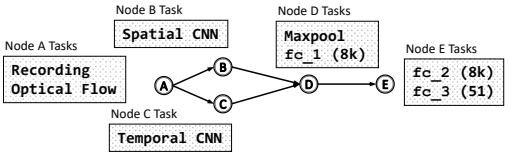
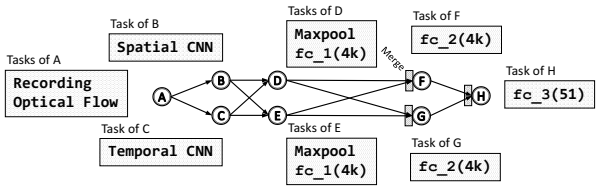


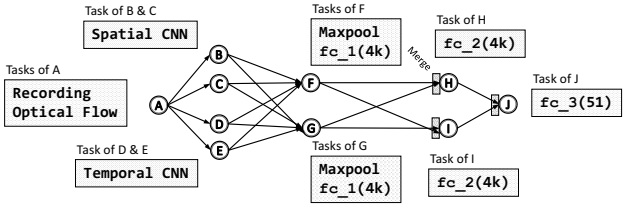
Fig. 10: (a) Loading time, (b) memory usage, (c) time per inference, and (d) energy per inference of general tasks in action recognition on a Raspberry Pi.



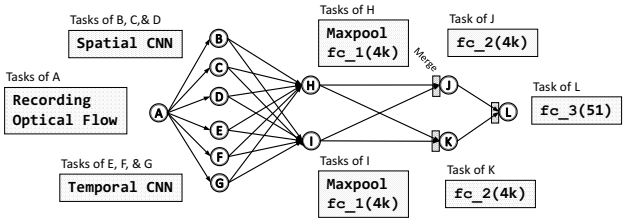
(a) Five robots: Exploiting model parallelism for `fc` layers.



(b) Eight robots: Exploiting model parallelism for *each* `fc` layer.



(c) 10 robots: Adding data parallelism for the two streams.



(d) 12 robots: Adding more data parallelism for the two streams.

Fig. 11: System architectures of action recognition.

size of which is dependent on the model and data parallelism of the previous robot and the input of the next robot. As discussed in the previous section, a single robot is unable to process data efficiently in real time. Hence, for demonstration, we perform distributed perception utilizing various systems, as shown in Figure 11, while measuring IPS, energy consumption, and end-to-end latency (Figures 12, 13, and 14, respectively)[1]. Our first system has five robots, Figure 11a, for which the final *fc* layers are distributed. Note that the systems with fewer than five robots are bounded by reloading time, and do not experience significant improvements in performance.

From eight robots, Figure 11b, our method performs model parallelism on both *fc* layers, creating two 4,096 `fc` layers per each layer. Furthermore, we are able to achieve 4.6× improvement in the performance and exceed the performance of TX2, shown in Figure 12. In the 10-robot system, two more robots process temporal and spatial streams exploiting data parallelism, illustrated in Figure 11c. New frames and optical flows are assigned in a round-robin fashion to two robots (of each stream) and are ordered using tags in subsequent robots. Finally, in the 12-robot system, more robots are assigned to process temporal and spatial streams with data parallelism. In summary, in comparison with a single robot, we gain up

[1]We evaluate these experiments and make the source code publicly available in this artifact [47].

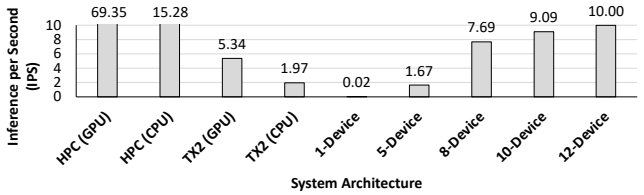


Fig. 12: Measured inference per second.

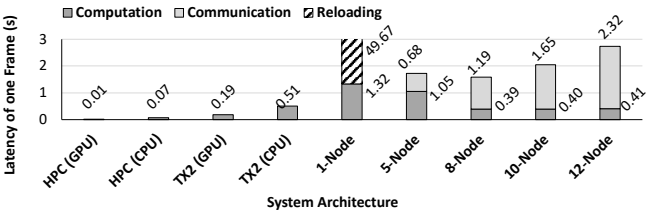
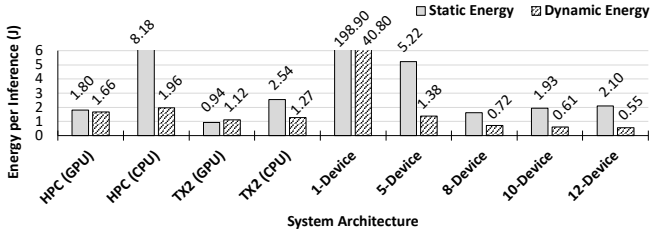


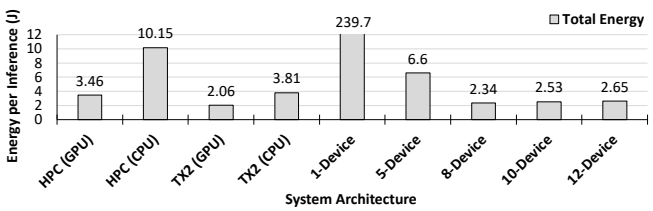
Fig. 13: Measured end-to-end latency of one frame.

to 90x energy savings and a speedup of 500x for IPS. As Figures 12 and 13 depict, although increasing the number of devices in a system also increases the latency notably, we observe a performance gain in IPS with a higher number of devices. This is because in both data and model parallelism, the systems hide latency by distributing or parallelizing tasks.

For the larger number of robots, we achieve not only similar energy consumption with TX2 but also save energy in comparison with the HPC machine. Figure 14b depicts that, except for the TX2 with GPU, the energy consumption per inference (i.e., $^{Watt}/_{performance}$) of systems with more than five robots is always better than in other cases (up to 4.3x and an average of 1.5x). Note that in our evaluations, the power consumption of the robot systems is inclined to higher energy consumption because (i) in comparison with TX2, since each robot's Raspberry-Pi is on a development board, it has several unnecessary peripherals, the energy consumption of which increases significantly with more robots, which is shown in static energy; (ii) TX2 is a low-power design with power gating capabilities that gates three cores if not needed, but robots do not have such capabilities; and (iii) the energy consumption of the robot systems also includes the energy for communication between the devices and the wasted energy of powering an idle core during data transmission.



(a) Measured static and dynamic energy consumption per inference.



(b) Measured total energy consumption per inference.

Fig. 14: Energy consumption per inference.

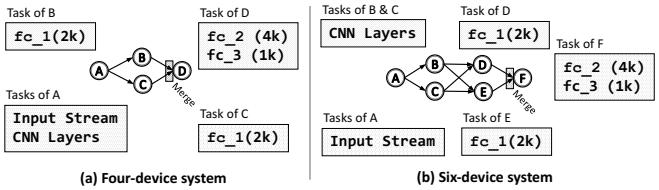


Fig. 15: System architectures for AlexNet.

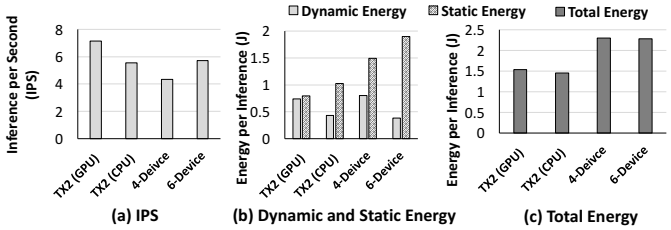


Fig. 16: AlexNet: Measured IPS (a), static and dynamic energy consumption (b), and total energy consumption (c).

C. Image Recognition

We apply our method to two popular image recognition models, described in §III. For AlexNet, Figures 15a and b display the generated tasks for four- and six-robot systems, respectively. While in the four-robot system, model parallelism is performed on the `fc_1` layer, in the six-robot system, additional data parallelism is performed on `conv` layers. We implement both systems and measure their performance and energy consumption, shown in Figure 16. Figure 16a depicts a performance increment by increasing the number of devices in a system. In fact, the achieved performance of the six-robot system is similar to the TX2 with CPU, and 30% worse than the TX2 with GPU. Furthermore, as discussed in the previous section, Figure 16b shows that most of the energy consumption of the Raspberry-Pi-based robots is because of the static energy consumption.

VGG16 (Figure 3b), in comparison with AlexNet, is more computationally intensive [38]. To distribute the model, our method divides the VGG16 model to several blocks of sequential `conv` layers. Figures 17a and 17b depict the outcome

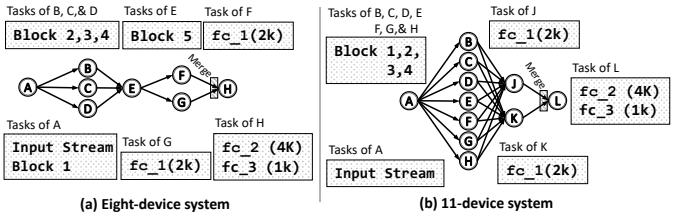


Fig. 17: System architectures for VGG16.

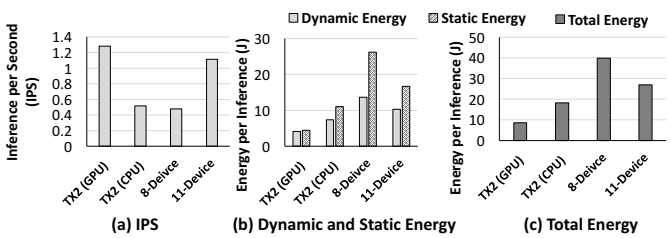


Fig. 18: VGG16: Measured IPS (a), static and dynamic energy consumption (b), and total energy consumption (c).

of task assignment for VGG16 with eight and 11 robots, respectively. Our method for `fc_1`, since its input size is large, performs model parallelism, while for `fc_2` and `fc_3`, since their computations are not a bottleneck, it assigns them to a single robot. We measure the performance and energy consumption of both systems and the TX2, shown in Figure 18. When the number of robots increases from eight to 11, we achieve 2.3x better performance by reassigning all `conve` blocks to a robot and performing more optimal data parallelism. In fact, compared to the TX2 with GPU, the 11-robot system achieves comparable IPS (15% degradation).

VII. Conclusion

In this paper, we proposed a technique to harvest the computational power of distributed robot systems by collaboration to enable efficient real-time recognition. Our technique uses model- and data-parallelism to effectively distribute computations of a DNN model among low-cost robots. We demonstrate our technique with a system consisting of Raspberry-Pi3-based robots by implementing a state-of-the art action recognition model and two well-known image recognition models. For future work, we plan to extend our work to heterogeneous robot systems and increase the robustness of our technique.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet Classification With Deep Convolutional Neural Networks," in *NIPS'12*. ACM, 2012, pp. 1097–1105.

[2] R. Collobert and J. Weston, "A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning," in *ICML'8*. ACM, 2008, pp. 160–167.

[3] D. Bahdanau, K. Cho, and Y. Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate," in *ICLR'15)*. ACM, 2015.

[4] M. S. Ryoo, K. Kim, and H. J. Yang, "Extreme Low Resolution Activity Recognition with Multi-Siamese Embedding Learning," in *AAAI'18*. IEEE, Feb. 2018.

[5] K. Simonyan and A. Zisserman, "Two-Stream Convolutional Networks for Action Recognition in Videos," in *NIPS'14*. ACM, 2014, pp. 568–576.

[6] Y. Wang, H. Li, and X. Li, "Re-Architecting the On-Chip Memory Sub-System of Machine-Learning Accelerator for Embedded Devices," in *ICCAD'16*, 2016, pp. 1–6.

[7] Y.-D. Kim, E. Park, S. Yoo, *et al.*, "Compression of Deep Convolutional Neural Networks for Fast and Low Power Mobile Applications," in *ICLR'16*. ACM, 2016.

[8] B. McDanel, S. Teerapittayanon, and H. Kung, "Embedded Binarized Neural Networks," in *EWSN'17*, 2017, pp. 168–173.

[9] S. Bang, J. Wang, Z. Li, *et al.*, "14.7 A 288μW Programmable Deep-Learning Processor with 270KB On-Chip Weight Storage Using Non-Uniform Memory Hierarchy for Mobile Intelligence," in *ISSCC'17*. IEEE, 2017, pp. 250–251.

[10] R. LiKamWa, Y. Hou, J. Gao, *et al.*, "RedEye: Analog ConvNet Image Sensor Architecture for Continuous Mobile Vision," in *ISCA'16*. ACM, 2016, pp. 255–266.

[11] Y. Kang, J. Hauswald, C. Gao, *et al.*, "Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge," in *ASPLOS'17*. ACM, 2017, pp. 615–629.

[12] J. Hauswald, T. Manville, Q. Zheng, *et al.*, "A Hybrid Approach to Offloading Mobile Image Classification," in *ICASSP'14*. IEEE, 2014, pp. 8375–8379.

[13] S. Teerapittayanon, B. McDanel, and H. Kung, "Distributed Deep Neural Networks Over the Cloud, the Edge and End Devices," in *ICDCS'17*. IEEE, 2017, pp. 328–339.

[14] Microsoft, "Embedded Learning Library (ELL)," https://microsoft.github.io/ELL/, 2017, [Online; accessed 11/10/17].

[15] M. Rastegari, V. Ordonez, J. Redmon, *et al.*, "XNOR-Net: Imagenet Classification Using Binary Convolutional Neural Networks," in *ECCV'16*. Springer, 2016, pp. 525–542.

[16] A. G. Howard, M. Zhu, B. Chen, *et al.*, "Mobilenets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *arXiv preprint arXiv:1704.04861*, 2017.

[17] S. Han, H. Shen, M. Philipose, *et al.*, "MCDNN: An Execution Framework for Deep Neural Networks on Resource-Constrained Devices," in *MobiSys'16*, 2016.

[18] Facebook, "Caffe2Go: Delivering real-time AI in the palm of your hand," https://code.facebook.com/posts/196146247499076/delivering-real-time-ai-in-the-palm-of-your-hand/, 2017, [Online; accessed 11/10/17].

[19] Google, "Introduction to TensorFlow Lite," https://www.tensorflow.org/mobile/tflite/, 2017, [Online; accessed 11/10/17].

[20] F. N. Iandola, S. Han, M. W. Moskewicz, *et al.*, "SqueezeNet: AlexNet-Level Accuracy with 50x Fewer Parameters and <0.5 MB Model Size," *arXiv preprint arXiv:1602.07360*, 2016.

[21] K. He, X. Zhang, S. Ren, *et al.*, "Deep Residual Learning for Image Recognition," in *CVPR'16*. IEEE, 2016, pp. 770–778.

[22] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in *ICLR'15*. ACM, 2015.

[23] C. Szegedy, W. Liu, Y. Jia, *et al.*, "Going Deeper with Convolutions," in *CVPR'15*. IEEE, 2015, pp. 1–9.

[24] D. Industries, "GoPiGo Robot," https://www.dexterindustries.com/gopigo3/, 2017, [Online; accessed 22/02/18].

[25] R. P. Foundation, "Raspberry Pi 3," https://www.raspberrypi.org/products/raspberry-pi-3-model-b/, 2017, [Online; accessed 11/10/17].

[26] R. P. Foundation, "Raspberry Pi 3," https://www.raspberrypi.org/products/camera-module-v2/, 2017, [Online; accessed 11/10/17].

[27] NVIDIA, "NVIDIA TK," http://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html, 2017, [Online; accessed 11/10/17].

[28] J. Mao, X. Chen, K. W. Nixon, *et al.*, "MoDNN: Local Distributed Mobile Computing System for Deep Neural Network," in *DATE'17*. IEEE, 2017, pp. 1396–1401.

[29] J. Dean, G. Corrado, R. Monga, *et al.*, "Large scale distributed deep networks," in *NIPS'12*. ACM, 2012, pp. 1223–1231.

[30] J. Redmon and A. Angelova, "Real-Time Grasp Detection Using Convolutional Neural Networks," in *ICRA'15*. IEEE, 2015.

[31] D. Maturana and S. Scherer, "VoxNet: A 3D Convolutional Neural Network for Real-Time Object Recognition," in *IROS'15*. IEEE, 2015.

[32] M. S. Ryoo, B. Rothrock, C. Fleming, *et al.*, "Privacy-Preserving Human Activity Recognition from Extreme Low Resolution." in *AAAI'17*. IEEE, 2017, pp. 4255–4262.

[33] C. Finn and S. Levine, "Deep Visual Foresight for Planning Robot Motion," in *ICRA'17*. IEEE, 2017.

[34] J. Lee and M. S. Ryoo, "Learning Robot Activities from First-Person Human Videos Using Convolutional Future Regression," in *IROS'17*. IEEE, 2017.

[35] S. Venkataramani, A. Ranjan, S. Banerjee, *et al.*, "Scaledeep: A scalable compute architecture for learning and evaluating deep networks," in *ISCA'17*. ACM, 2017, pp. 13–26.

[36] O. Russakovsky, J. Deng, H. Su, *et al.*, "Imagenet Large Scale Visual Recognition Challenge," *IJCV*, vol. 115, no. 3, pp. 211–252, 2015.

[37] K. He, X. Zhang, S. Ren, *et al.*, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on Imagenet Classification," in *ICCV'15*. IEEE, 2015, pp. 1026–1034.

[38] A. Canziani, A. Paszke, and E. Culurciello, "An Analysis of Deep Neural Network Models for Practical Applications," *arXiv preprint arXiv:1605.07678*, 2016.

[39] J. Choi, W. J. Jeon, and S.-C. Lee, "Spatio-Temporal Pyramid Matching for Sports Videos," in *ICMR'8*. ACM, 2008, pp. 291–297.

[40] G. Farnebäck, "Two-Frame Motion Estimation Based on Polynomial Expansion." Springer, 2003, pp. 363–370.

[41] H. Kuehne, H. Jhuang, E. Garrote, *et al.*, "HMDB: A Large Video Database for Human Motion Recognition," in *ICCV'11*. IEEE, 2011, pp. 2556–2563.

[42] A. Coates, B. Huval, T. Wang, *et al.*, "Deep Learning with COTS HPC Systems," in *ICML'13*. ACM, 2013, pp. 1337–1345.

[43] T. A. S. Foundation, "Apache Avro," https://avro.apache.org, 2017, [Online; accessed 11/10/17].

[44] NVIDIA, "NVIDIA Jetson TX," http://www.nvidia.com/object/embedded-systems-dev-kits-modules.html, 2017, [Online; accessed 11/10/17].

[45] F. Chollet *et al.*, "Keras," https://github.com/fchollet/keras, 2015.

[46] M. Abadi *et al.*, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[47] R. Hadidi, J. Cao, M. Woodward, *et al.*, "Real-time image recognition using collaborative iot devices," in *ReQuEST '18*. ACM, 2018.