Distributed and Configurable Architecture for Neuromorphic Applications on Heterogeneous Cluster

Khadeer Ahmed, Qinru Qiu
Department of Electrical Engineering and Computer Science, Syracuse University, NY 13244, USA
Email {khahmed, qiqiu} @syr.edu

Mangesh Tamhankar
Intel Corporation
Santa Clara, CA USA
Email mangesh.tamhankar@intel.com

Abstract—**With the proliferation of application specific accelerators, the use of heterogeneous clusters is rapidly increasing. Consisting of processors with different architectures, a heterogeneous cluster aims at providing different performance and cost tradeoffs for different types of workloads. In order to achieve peak performance, software running on heterogeneous cluster needs to be designed carefully to provide enough flexibility to explore its variety. We propose a design methodology to modularize complex software applications with data dependencies. The software application designed in this way have the flexibility to be reconfigured for different hardware platforms to facilitate resource management, and features high scalability and parallelism. Using a neuromorphic application as a case study, we present the concept of modularization and discuss the management, scheduling and communication of the modules. We also present experimental results demonstrating the improvements and effects of system scaling on throughput.**

Keywords—Distributed computing; structure based scheduling; heterogeneous computing; pipelining; latency hiding; modularization

I. INTRODUCTION

Modern computing systems are increasingly becoming more heterogeneous. This is due to a wide variety of computing architectures and accelerators such as multi-core CPU, GPU, FPGA, etc. being used. Different architectures provide different performance and cost tradeoffs, which greatly extends software design space. If utilized properly, they can significantly optimize the performance of software systems. The heterogeneity is most beneficial to complex software systems, which involve large quantities of data processing at multiple levels and different modalities. Some of such domains are scientific computing, big data, machine learning, neuromorphic applications, financial modeling etc. These applications consist of different types of workload, which can benefit from the diversified architectures.

There are many challenges towards achieving full potential of a heterogeneous computing cluster. Program designer needs to pay attention to resource mapping, utilization, task scheduling, etc. The same software working efficiently on one heterogeneous cluster may not have high hardware utilization on another. This work investigates design methodology for developing software applications with high flexibility so that they can be reconfigured to fit to different heterogeneous clusters. Our goal is to improve the reusability and portability of distributed software system, and our solution is through modularization and standard inter-module communication architecture.

This work is partially supported by the National Science Foundation under Grant CCF-1337300 and Intel Corporation under CG #20263363.

Today the industry is focusing towards brain inspired computing due to its efficiency, scalability and its ability to solve complex problems. Perception in biological system involves different type of processing in different brain area. For example, pattern matching happens in the basal ganglia where brain processes large stimuli quickly in a parallel fashion. For more sophisticated processing such as reasoning, relatively slower sequential processes will occur in the sensory association cortex. Like the brain, neuromorphic applications have different computing requirements for different stages of processing. Therefore, it will benefit from heterogeneous computing clusters.

In this work, we use one such neuromorphic application, Intelligent Text Recognition System (ITRS) [1], as a case study to discuss the design methodology of our distributed software architecture. We encourage modularized design and use of multiple programming paradigms, such as multi-threaded programming, CUDA etc., to achieve best possible optimization for each module in the distributed software system. The design enables flexible configurations to ensure best possible throughput at the module level for the intended hardware platform. We propose the use of message passing interface (MPI) [2] industry standard for inter-module communication and novel structure based scheduling for scalable implementation. The module functionality is decoupled from its rank identifier; therefore, the system functionality is solely dependent on the distributed architecture. The main contributions of our work are as follows.

1. Uniform communication architecture with point-to-point links is introduced as a foundation, which enables modularization of complex systems. The communication architecture scales in a distributed way employing asynchronous communication with multiple modules.
2. Design methodology is presented to pipeline modules with data dependency for out-of-order workload execution. Distributed pipeline control is used for asynchronous processing.
3. Novel structure based scheduling is introduced for achieving maximum performance for asynchronous workload processing with varying module latencies.
4. A methodology for flexible scaling of the modular software system is proposed to achieve desired throughput by leveraging different hardware resources in a heterogeneous cluster.

We perform several experiments to show the speed up gained over the non-reconfigurable implementation and provide several test case results involving bottleneck identification and removal using the scalable architecture, which would not have been possible without modularization and reconfiguration. We also demonstrate the flexibility of the proposed methodology in running the application in resource constrained situations.

The rest of the paper is organized as the following. Section II discusses previous works in distributed software design methodology. In Section III we will discuss the background of neuromorphic algorithms used in ITRS and the existing ITRS architecture. In Section IV, V and VI we elaborate on the modularization methodology, which includes the communication architecture, the methodology for pipelined execution of distributed applications, and the novel structure based scheduling. Finally, in Section VI, the experimental results are discussed followed by conclusions.

II. Related Work

Distributed systems are widely used and have been extensively studied. Different kinds of distributed pipeline based architectures are proposed. A state based distributed pipeline framework is presented in [3]. Here the compute nodes are separated from the pipeline control. Instead of message passing the state objects are passed which encapsulate the data. The load balancing is achieved through producer/consumer relationship i.e. processing happens asynchronously. However, there is an extra overhead in creating and decoding state objects at every stage apart from data processing. A distributed pipeline processing architecture composed of flow-models, called meta-pipeline is proposed for general-purpose computation [4]. The architecture is suitable for stream based processing. This requires input and output streams along with the parameters for every flow-model. These details and other properties are encapsulated in XML. This kind of modularization enables distributed task based execution. Though this system is distributed it requires centralized management to assign and load flow-models.

Fully utilizing the performance of heterogeneous resources is a challenging task. Design methodology for executing applications on heterogeneous platforms, which are specified as synchronous dataflow (SDF) graphs is proposed in [5]. The authors try to maximize the end-to-end throughput of an application developed in OpenCL by modeling it using SDF graph. In contrast to the aforementioned related works, the goal of our work is to show how complex application with various processing requirements can be converted to scalable, distributed applications. We present a scheme through which the data dependency is maintained by utilizing a pipeline, at the same time allowing out-of-order execution of workloads for higher hardware utilization. Also the concept of structure based scheduling is proposed with distributed flow control on a heterogeneous cluster.

III. Background of Intelligent Text Recognization

The neuromorphic model adopted by the ITRS software is mainly built based on the Brain-State-in-a-Box (BSB) attractor model [6] [7] [8] and the Cogent Confabulation model [9]. The BSB models provide matching patterns for each character image. The cogent confabulation algorithms combine information from the BSB model to form more complex objects such as words or sentences. During this procedure, it suppresses the inputs that do not have strong association with others and enhances the remaining inputs. In other words, the confabulation model eliminates those BSB results that do not form meaningful words and sentences. Therefore, ITRS is capable of extracting meaningful text from noisy and occluded document images. The salient feature of ITRS is that it provides contextually correct sentence reconstruction even if there are illegible characters or words in the document image [1]. This is enabled by a trained knowledge base, which captures the statistical information among building components in English language, from letters and words to phrases and part-of-speech tagging.

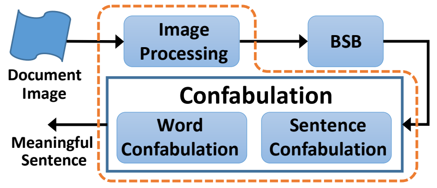


Fig. 1. ITRS pipeline

The information processing in ITRS has several stages, which can be arranged as a pipeline shown in Fig. 1. After simple image processing which corrects image distortion, skew and warping, the character images are segmented from document image and forwarded to BSB, where fuzzy pattern matching is performed. The pattern matching results will be processed by word and sentence level confabulation [10] for inference based information association and error correction. The pattern-matching layer (BSB) is trained on clean font image. The word level confabulation is trained by reading a dictionary and the sentence level knowledge base is trained by reading multiple classic literatures.

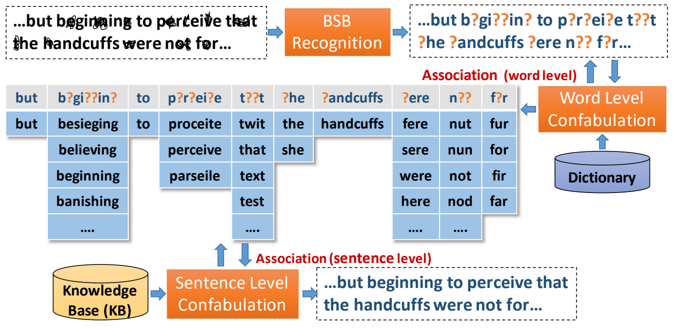


Fig. 2. ITRS cognitive model

The cognitive model of ITRS is illustrated by an example in Fig. 2. Given a noisy document image, the BSB provides pattern-matching candidates for each character image using best effort. Each question mark in the figure represents all 26 possible alphabets. The word confabulation layer builds word candidates while filtering out any meaningless words and the sentence confabulation layer selects the words that form the most meaningful sentence. It is easy to see that, for each sentence, one sentence confabulation task and multiple word confabulation tasks must be executed, along with even more number of BSB pattern matching tasks. The computation tasks within the same layer are independent to each other and hence can be implemented in parallel.

Based on the discussion above it is clear that there are different distinct stages in the pipeline with different compute requirements. Though the original software architecture of ITRS proposed in [1] is parallel, it is not properly modularized and does not scale efficiently. The original ITRS has only two modules. One of them handles BSB pattern matching, and the other handles all the rest of computation tasks. The task division among the two modules is shown by dotted line in Fig. 1. Since image processing applies different algorithms for processing different regions of the noisy image, it is a thread intensive task where each thread performs small but distinct computation. Therefore, each type of computation gets a thread pool for efficient asynchronous computation. BSB is an attractor model of auto associative memory, it is compute intensive and best suited for high performance accelerators. Word and sentence confabulation tasks perform sparse computation and have intensive random memory access. Such variety in workload characteristic is common in full-scale neuromorphic applications. Therefore, we propose an efficient, scalable architecture for improving the throughput by employing pipelining concepts and out of order computing for such systems. These systems have inherent data dependency, so we also propose a novel structure based scheduling methodology to efficiently utilize the compute resources in a distributed manor with inherent load balancing capabilities. Though we have selected ITRS as a case study for this work the methodology can be extended to other complex software applications where computation can be performed in isolated stages thus forming an asynchronous pipeline.

IV. Modularization of Software Architecture

A. *Balanced pipeline processing*

Without loss of generality, we consider a target heterogeneous cluster with Intel Xeon CPU nodes, Intel Xeon Phi nodes and NVIDIA GPGPU nodes. For an efficient heterogeneous cluster implementation, we break the ITRS functional pipeline into stages. Since each stage performs a very specific task it is also referred to as a layer, for example image layer, BSB layer etc.

In order to achieve high performance, all pipeline stages should have the same throughput. However, the workloads of different layers differ significantly. For example, each document page is typically 2500x3300 pixel image. The number of characters on a page is in the range of 10^3, the number of words on a page is in the range of 10^2, and the number of sentences is even less. The architecture should provide the flexibility so that a stage with heavy load can grab more computing resources.

In the proposed architecture, a pipeline stage consists of multiple identical software processes called modules. Each module works independently on a sequence of incoming jobs. Each job is a small unit of workload in that particular layer, such as performing pattern recognition on character images, constructing one word, etc. Each software module which runs on a hardware node employs multi-threading to achieve maximum efficiency. The number of threads in a module and the number of modules in a layer are configurable and they are determined to keep a balanced pipeline. In order to allocate more resources to a particular layer, we simply need to instantiate more modules or threads in a module of that layer. In a heterogeneous system, their selection does not only depend on the workload but also the hardware that the modules and threads are running on.

The modules use MPI for inter process communication. Typically, MPI based programs are developed to benefit from parallel processing on distributed or shared memory computer clusters. However, this is not the only programming model supported by MPI standard. Each stage of the proposed pipeline performs very distinct task and can run on a variety of hardware platforms in a cluster. Hence we use MPI for packetized data communication, which will be discussed later. The data flows through the pipeline for stream processing. Each data message specifies a job to be processed in its destination layer. Whenever a thread in a module is free, it fetches an incoming job from a common receiving queue and starts working on it. Therefore, the workload inside a module is distributed among all thread in a balanced way.

The new modularized pipeline is shown in Fig. 3. Hence forth in the paper, this will be referred to as ITRS pipeline. With the new pipeline architecture each stage is independently scalable. Within a pipeline stage, all workloads are processed independently and their completion is out-of-order. The result gather module is introduced as the fifth stage to assemble the output of sentence confabulation results to in-order output.

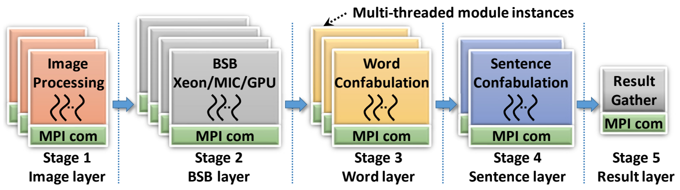


Fig. 3. Modular ITRS pipeline

The above modularization and parallelization method enable us to separate individual modules and run them on most appropriate hardware platform that fits their computation requirement. Hence it is capable of effectively utilizing resources in a heterogeneous cluster. The proposed software architecture provides many knobs to fine tune the performance. For example, users can adjust the hardware utilization and system performance by selecting the number of threads in each module, the number of modules in each layer, and specifying CPU thread affinity.

B. *Uniform inter-module communication*

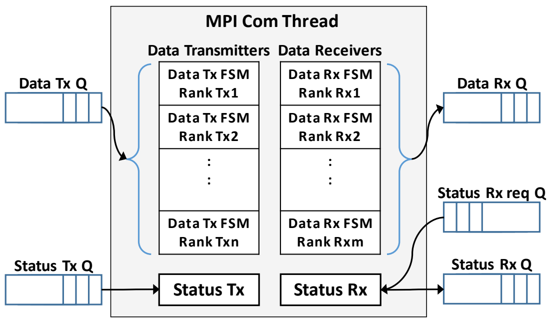


Fig. 4. MPI communication sub-module architecture

The MPI communication sub-module (MCSM) which is a reusable library, is designed to interface in a thread safe way with all modules in the pipeline. It is a key enabler for reliable and scalable implementation. MCSM is always attached to a software module, which is also referred as its *parent module*. Each functional module can only have one MCSM.

MCSM contains only one thread and all messages are funneled through this thread. It can send and receive data messages from multiple ranks simultaneously. Apart from data messages, it also supports sending and receiving status messages to and from multiple ranks to facilitate inter-module synchronization. All status messages are one byte integer messages and data messages are given size character messages. While non-blocking MPI communication is used for data transmission and status receiving; blocking MPI communication is used for status transmit. In this way we can guarantee that the status is sent in order.

The architecture of this sub-module is shown in Fig. 4. MCSM manages five thread safe blocking queues. As shown in the figure, they are for data send (Data Tx Q), data receive (Data Rx Q), status send (Status Tx Q) and status receive (Status Rx req Q and Status Rx Q). The parent module is the producer for Data Tx Q, Status Tx Q, and status Rx req Q. It is the consumer of Data Rx Q and Status Rx Q. MCSM doesn't block on any of these queues to maintain reliable, always open communication. However, depending on the computing requirements of the threads in the parent module these queues can be used in a thread non-blocking or thread blocking manner based on the full and empty flags of the queues. As we mentioned before, each received data message specifies a job to be processed. If the parent module consists of multiple threads, all threads fetch jobs from Data Rx Q whenever they are free.

MCSM maintains two types of data objects; Data Transmitters and Data Receivers. It creates one data transmitter or data receiver object for each MPI rank it needs to communicate with. Each of these objects contains a state machine to manage the communication with that particular rank.

The data messages generated by the parent module also consist of destination rank information which are forwarded to MCSM through Data Tx Q. Each Data Transmitter object is interfaced with a local queue. These queues are stored in a map container with the destination rank of the associated object as the key. Whenever there is data available on the Data Tx Q it is copied on to appropriate local queue with the same key. Transmitter pools are created by sharing the local queues among select data transmitter objects. The data messages in the local queue can be sent to any rank in that pool as long as it is free to receive. Therefore, these transmitter pools enable load balancing in a distributed way at module level among the ranks in the pool. The data receiver objects en-queue all received data messages to Data Rx Q. It is the responsibility of the consumer of this queue to de-queue data messages from them.

MCSM also maintains one object for the status transmitter and receiver, and it is shared among all ranks. As we mentioned before, blocking MPI communication is performed for status information. The status communication is only used during initialization and termination.

The Status Tx Q receives status messages from the parent module along with the destination rank to MCSM. The Status Tx object sends one message at a time by initiating a blocking MPI send. To request the status message from a particular module, the parent module sends the rank of the target module to Status Rx Req Q. This initiates a non-blocking MPI receive. Another status request is not processed until current one is complete. The received status message is en-queued to Status Rx Q. We make the receive operation non-blocking and give power to the parent module to decide whether to block on an empty Status Rx Q or not.

The MPI communication thread keeps on polling the TX Q and Status Req Q, hence it is always busy from creation to termination. The thread round robins on four functions: data receive, status receive, data transmit and status transmit. The detailed communication protocol is discussed next.

C. Communication Protocol

All communication is point-to-point and follow a flow control based communication protocol. The receiver has a capacity limit, which is the maximum length of Data Rx Q. If there is room for data in the queue then the receiver is inferred to be in ready state else it is inferred to be in busy state. In this protocol the transmitter sends a request message to the receiver. Based on the state of the receiver there can be two cases as shown in Fig. 5. If the receiver is ready, then it sends a positive acknowledgement to the transmitter, otherwise it sends a negative acknowledgement . Once the receiver becomes ready, it will again send a positive acknowledgement message. The transmitter sends data only after receiving a positive acknowledgement. If a negative acknowledge is received, it will yield its turn to other transmitter objects, in this way load balance is achieved. These protocols are implemented as state machines on transmitter side and receiver side. The description of the state machines is given below.

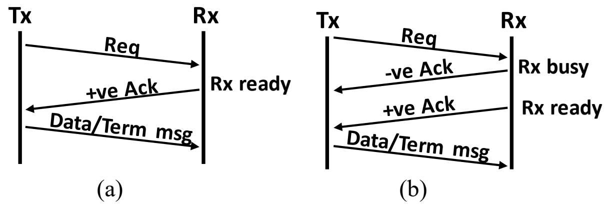


Fig. 5. Flow Control Protocol. (a) Receiver is ready (b) receiver is busy

The Transmitter State Machine (TSM) has six states as shown in Fig. 6. The TSM starts in send request state (SEND_REQ) where, a request message is sent to the receiver and the state transitions to wait request acknowledgement (WAIT_REQ_ACK) state. TSM now waits for acknowledgement message from the receiver. If it receives a positive acknowledgement, then it transitions to initiate send message (INIT_SEND_MSG) state else it transitions to initiate ready acknowledgement state (INIT_READY_ACK) where a non-blocking MPI receive is called and the state transitions to wait ready acknowledgement (WAIT_READY_ACK). After receiving an acknowledgement, the TSM transitions to INIT_SEND_MSG where a non-blocking MPI send is called and the state transitions to wait send message (WAIT_SEND_MSG) where it waits till the MPI message is sent. Then it transitions to SEND_REQ and the process starts all over.

The receiver state machine (RSM) also has 6 states as shown in Fig. 7. The RSM starts in initiate receive request state (INIT_REC_REQ) where, a MPI non-blocking receive is called and the state transitions to wait request (WAIT_REQ) state.

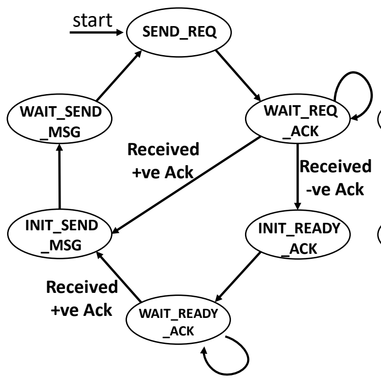


Fig. 6. Transmitter State Machine

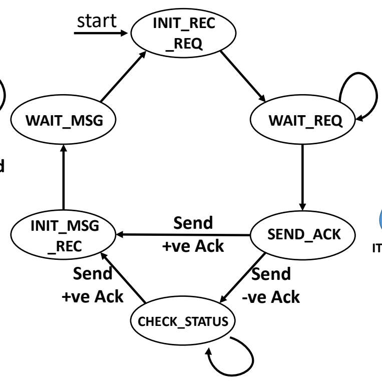


Fig. 7. Receiver State Machine

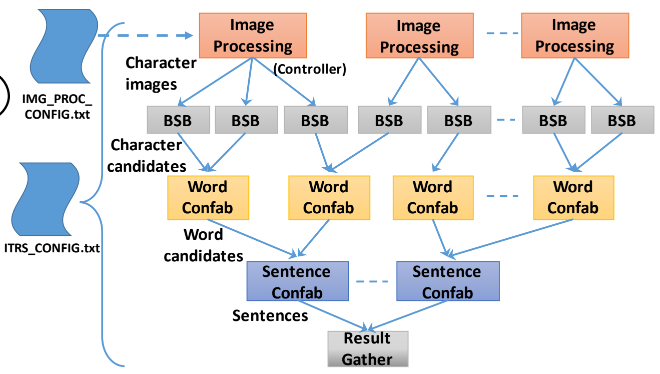


Fig. 8. Typical, scalable ITRS system topology

RSM now waits for a request from the transmitter. Upon receiving the request, it transitions to send acknowledgement (SEND_ACK) state. In SEND_ACK state the RSM checks if Data Rx Q is full and sends a positive acknowledgement if there is room for data then the state transitions to initiate message receive (INIT_MSG_REC) state. Otherwise, if the queue is full then a negative acknowledgement is sent and the RSM transitions to check status (CHECK_STATUS) state. In this state the RSM monitors the Data Rx Q size, as soon as the parent module de-queues a data message the RSM sends a positive acknowledgement and the state transitions to INIT_MSG_REC state. In INIT_MSG_REC state a non-blocking MPI receive is called and state transitions to wait message (WAIT_MSG) where RSM waits till message is received. After receiving the message, state transitions to INIT_REC_REQ and the process starts over.

The TSM (RSM) will make one transition according to the diagram when the MCSM enters the data transmit (data receive) function in the round robin process.

V. Scalable ITRS structure

Each layer in the ITRS has a set of independent modules with the attached MPI communication sub-module. The modules in adjacent layers have point-to-point communication links. A typical, scalable ITRS system configuration is shown in Fig. 8.

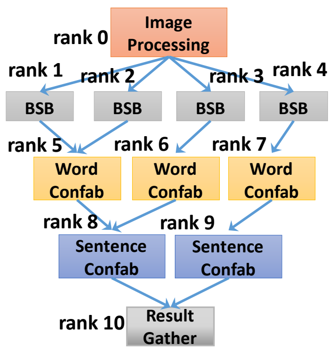
In this structure the data flow is restricted through point-to-point communication between modules, hence data flow is distributed and not routed through centralized communication hubs. Since the architecture is pipelined, the neighboring modules can be allocated to physically nearby compute nodes to reduce communication latency. The workloads across the system are processed in parallel and may complete out-of-order as all the modules in the system are running independently and asynchronously. The data communication between modules is handled by non-blocking MPI communication, which allows communication and computation to happen in parallel. Therefore, the communication latency is hidden.

We use a configuration file, ITRS_CONFIG.txt, to specify the system topology and tunable parameters, such as the number of modules and thread in each layer and the CPU affinity of modules. One of the image processing module in the system behaves like a controller which synchronizes the initialization and termination of all modules in the system. This topology enables scaling by adding more modules to each layer and thereby increasing that layer's throughput. Therefore, each module tries to scale at the multi-core level and the overall system architecture enables scaling at the cluster level.

VI. Structure based scheduling

The proposed methodology for achieving scalable throughput is flexible due to out of order computation, but at the same time poses some challenges for scheduling the workloads. This is due to the dependency between layers and among the data. For example, a word cannot be processed by the word confabulation module until all the character candidates of that word are processed by the BSB layer and the results of these candidates must be sent to the same word confabulation module. Similar situation can be found in the sentence layer, the processing cannot starts until all word candidates are processed and their results are sent to the same sentence module. In other words, we need to make sure that the data belonging to the same workload hierarchy are able to reach the same processing module.

We solve this problem by designing the communication network topology of the distributed system according to the structure of its workload. Using ITRS as an example, as seen in Fig. 8 except the image processing module all the other modules have single output which converge to the inputs of downstream modules. The topology defines a set of upstream fan-ins for each module. As long as higher layer workloads are assigned to those fan-in modules, we can guarantee that the downstream workload generated by them will converge to the same module. For example, we know that characters assigned to BSB module 1, 2, and 3 will all go to the same sentence module by simply following the path in the network. In this way, we avoided attaching tags



(a)

Paths	BSB Ranks	Word Ranks	Sentence Ranks
A-1-1	1	5	8
A-1-2	2	5	8
A-2-1	3	6	8
B-1-1	4	7	9

(b)

Fig. 9. Structure based scheduling (a) Example topology (b) Sentence paths for image rank 0

to data message to specify its affinity or complicated routing algorithm.

As an example, let us consider a topology with the MPI ranks as shown in Fig. 9 (a). We can divide the BSB modules into two sets, modules 1, 2, and 3 converging to sentence module 8, and module 4 converges to sentence module 9. We call this as level-1 grouping of BSB. We can further divide the first set of BSB modules into two sets, {BSB1, BSB2} and {BSB3} based on their connectivity to the word confab module. We call this level-2 grouping. Based on these groupings four possible paths can be identified for scheduling workloads to BSB modules as shown in Fig. 9 (b). All characters belonging to same sentence can be assigned paths {A-1-1, A-1-2, A-2-1} or {B-1-1}, which belong to level-1 grouping. All characters belonging to same words can be assigned paths from level-2 grouping, {A-1-1, A-1-2} or {A-2-1} for one sentence or {B-1-1} for another sentence.

If a module is congested, because of flow control, all its upstream modules will get congested. For example, if word confab module 5 is congested, then all BSBs whose level 2 grouping is associated to this module will eventually congest. Therefore, the image processing module will assign the workload to BSBs belonging to other level 2 grouping. Please note that all modules support out-of-order execution. Even if one of the module is congested, the other modules in the same layer keeps sending new completed results to the downstream module. For example, even if word module 5 is congested, word module 6 will still keep the sentence confab module 8 busy. In this way we achieve workload balance at module level. All modules construct a job by obtaining its elements from one or many data messages received asynchronously from upstream modules. Partially constructed jobs do not block computation resources. Therefore, deadlocks due to data dependency cannot occur in this architecture.

VII. Experiments

Our experimental setup involves 3 Xeon multi socket nodes, NVIDIA GPU (C2075) and one Xeon Phi card (5110P). The node and accelerator details are given in TABLE I and TABLE II respectively. The ITRS topology is layer based which mimic the stages of a pipeline. For an efficient pipeline, it is necessary for all stages to have equal throughput or downstream stage has higher throughput so that there are no bottlenecks. In the ITRS system the BSB layer is the most computation intensive and will cause the primary bottleneck. Therefore, we test only meaningful configurations, which allocate resource in a way that eliminates bottlenecks. The select configurations are shown in TABLE III. In this table the different nodes and accelerators assigned for every layer in each configuration is shown.

We run several experiments with specified configurations to demonstrate the effectiveness of scaling and pipelining. Fig. 10 shows the comparison of throughput (measured by runtime per document image) of every configuration listed in TABLE III. Each configuration is run on 1, 4 and 16 document images. We can see that due to pipelining the throughput increases as the amount of independent workload increases. In configuration C2 accelerator A2 is used as an independent node hence it is capable of running independent binaries. Since one instance of BSB used partial resources on A2 we assigned two instances of BSB to run on A2. In general, it is apparent that as the amount of resources increases, the overall run time reduces.

To demonstrate the effect of distributed load balancing we turned off rank pooling for configuration C4 and the runtime increased to 67.7s for processing one image. This is because, among the two BSB nodes one is a Xeon node which has much lower computing power compared to GPU. Since load balancing is turned off the workloads were uniformly scheduled among them hence resulting in increased processing time. The flow control protocol is critical for asynchronous distributed architectures. We found that the experiments employing no flow control resulted in poor reliability as the downstream modules were prone to buffer overflow errors. Due to structure based scheduling no extra effort was required to maintain accuracy of results with changing topology across various configurations.

TABLE I. Node details

CPU Node	Logical Cores	Frequency (GHz)	Total Cache (MB)	RAM (GB)
N1	16	3.2	16	47
N2	12	2.1	15	32
N3	32	2.9	40	189

TABLE II. Acclerator details

Accelerator Node	Card	Frequency (GHz)	RAM (GB)
A1	C2075	1.15	6
A2	5110P	1	8

TABLE III. Configurations

Cnfg. No.	Image	BSB	Word	Sentence	Result
C1	N2	A2	N2	N2	N2
C2	N2	A2, A2	N2	N2	N2
C3	N1	A1	N1	N1	N1
C4	N1	N3, A1	N1	N1	N1

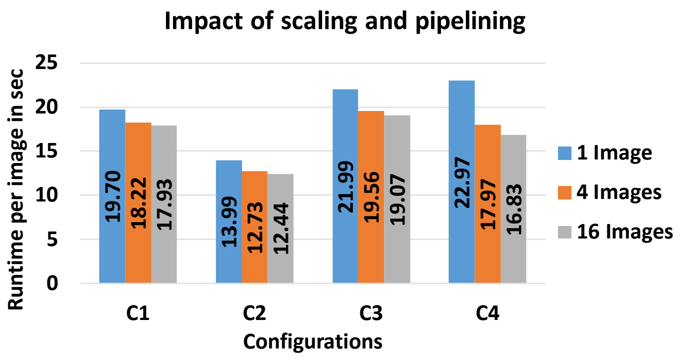


Fig. 10. Runtime values for different configurations

VIII. Conclusion

Through this work we have shown that data dependent applications with variety of different workload processing requirements can be implemented as pipelined and distributed systems on a heterogeneous cluster. We also proposed a structure based scheduling scheme to enable seamless scaling and provide module level load balancing in a non-centralized way. Hence achieving maximum resource utilization and providing best throughput for available hardware resources.

References

[1] Q. Qiu, Q. Wu, M. Bishop, R. E. Pino and R. W. Linderman, "A Parallel Neuromorphic Text Recognition System and Its Implementation on a

Heterogeneous High-Performance Computing Cluster," *IEEE Transactions on Computers,* vol. 62, no. 5, pp. 886-899, May 2013.

[2] *http://www.mpich.org/.*

[3] G. Z. Sun and G. Chen, "Distributed Pipeline Programming Framework for State-Based Pattern," in *2009 Eighth International Conference on Grid and Cooperative Computing*, 2009.

[4] S. Yamagiwa, L. Sousa and T. Brandao, "Meta-Pipeline: A New Execution Mechanism for Distributed Pipeline Processing," in *Parallel and Distributed Computing, 2007. ISPDC '07. Sixth International Symposium on*, 2007.

[5] L. Schor, A. Tretter, T. Scherer and L. Thiele, "Exploiting the parallelism of heterogeneous systems using dataflow graphs on top of OpenCL," in *The 11th IEEE Symposium on Embedded Systems for Real-time Multimedia*, 2013.

[6] J. Park and Y. Park, "An optimization approach to design of generalized BSB neural associative memories," *Neural computation,* vol. 12, no. 6, pp. 1449-1462, 2000.

[7] Y. Park, "Optimal and robust design of brain-state-in-a-box neural associative memories," *Neural Networks,* vol. 23, no. 2, pp. 210-218, 2010.

[8] A. Schultz, "Collective recall via the Brain-State-in-a-Box network," *Neural Networks, IEEE Transactions on,* vol. 4, no. 4, pp. 580-587, 1993.

[9] R. Hecht-Nielsen, Confabulation theory: the mechanism of thought, Springer Heidelberg, 2007.

[10] Q. Qiu, Q. Wu, D. J. Burns, M. J. Moore, R. E. Pino, M. Bishop and R. W. Linderman, "Confabulation based sentence completion for machine reading," in *Computational Intelligence, Cognitive Algorithms, Mind, and Brain (CCMB), 2011 IEEE Symposium on*, 2011.