2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications

A parallel runtime framework for communication intensive stream applications

Servesh Muralidharan†, Kevin Casey§ and David Gregg†
†*Lero and School of Computer Science and Statistics, Trinity College Dublin, Dublin 2, Ireland*
Email: muralis@scss.tcd.ie, David.Gregg@cs.tcd.ie
§*Lero and School of Computing, Dublin City University, Glasnevin, Dublin 9, Ireland*
Email: Kevin.Casey@computing.dcu.ie

***Abstract*—Stream applications are often limited in their performance by their underlying communication system. A typical implementation relies on the operating system to handle the majority of network operations. In such cases, the communication stack, which was not designed to handle tremendous amounts of data, acts as a bottleneck and restricts the performance of the application.**

In this paper, we propose a parallel runtime framework that integrates the communication operations with stream applications, and provides a common parallel processing engine that can execute both the communication and computation operations in parallel on multicore processors. We place an emphasis on the low-level details required to implement such a framework, but also provide some guidelines on how an application programmer can employ the framework.

Our runtime system uses a set of operations represented as filters to perform the relevant computations on the data stream. Filters that handle the application specific operations are categorized as computation filters and those that transform data to and from network devices are classified as communication filters. Computation filters are designed by the user and are specific to the application. Communication filters are provided by the runtime system and are built using system software that allows direct access to network hardware. Such system software allows the network operations to be performed by the runtime system in parallel, leading to better communication performance.

Applications that are designed for this framework are built by constructing application specific computation filters and then connecting them to the communication filters provided by the runtime system. This abstracts the low-level programming of network adapters and protocols by the application developer, making it easier to build stream applications that take advantage of the improved communication performance. Moreover, by dynamically replicating and statically scheduling such filters on the given multicore architecture, it is possible for the runtime system to process multiple data streams in parallel.

We are able to parallelize stream applications and achieve speedups of more than a factor of eight in all the applications we tested. The results show that our system scales to as many parallel processes as there are cores on our computer, and achieves speedups of more than a factor of ten in some cases compared to sequential implementations.

***Keywords*-Streaming model, data parallelism, clusters.**

I. Motivation

Applications that follow a streaming model [1]–[4] are expressed as a network of *filters* connected by communication *channels*. Tokens of data flow along the communication channels between filters. Each filter consumes one or more *streams* of data from its input channels, and produces streams of data on its ouput channels. For applications that fit the model well, stream processing can greatly reduce the working memory requirements. Since the data flows through a sequence of filters in a pipelined style, it is often possible to operate on a relatively small window of the data at any given time, thereby making it possible to operate on conceptually infinite data streams.

Traditionally, the stream processing model was used primarily for signal processing type applications, where the application processes a continuous stream of signal inputs. For example, many video processing applications are modeled as a set of image filters operating on a stream of video frames, where the entire video may be hundreds of gigabytes in size, but the processing can be done with just a few frames at a time. However, in more recent years it has been found that the stream model is highly suitable for many applications that process very large data sets, in the order of terabytes or even larger. Applications such as financial trading systems [5], database systems [6], network data analysis [7], [8] are some examples that follow the streaming model and also operate on very large data sets.

Stream processing is highly suited to multicore and parallel computing, because the filters can typically execute in parallel. Smaller stream applications often achieve high levels of parallelism on multicore machines. However, as the stream processing model is used to address larger data processing problems, more computational resources are needed, and a common solution is to distribute the filters across a number of machines. In such cases, the applications are typically limited by the traditional communication mechanisms [9]–[13]. The main reason is that these systems have limited support for operating on parallel streams and are burdened by abstraction layers that provide compatibility for non-streaming applications.

Several high speed networks [14] exist that could handle such communication requirements but these require expensive hardware and specialized programming skills. Commodity network adapters based on 10-gigabit ethernet (10GbE) are a comparatively cheaper alternative and are being widely deployed in clusters and grids in data centers. Previous research [15], [16] has been done on optimiz-

978-0-7695-5022-0/13 $26.00 © 2013 IEEE
DOI 10.1109/TrustCom.2013.142

IEEE computer society

ing ethernet for improving communication in such cases. However, they do not use many of the features provided by modern day network adapters such as hardware queues, flow control, etc., Also, they do not focus specifically on streaming applications, making it difficult to utilize them properly.

II. Contribution

In this paper, we propose a parallel runtime framework for building stream applications that are bound by very large communication. A simple framework is used, where the filters are described as *actors* and the application is represented by connecting such *actors* into a directed acyclic graph (DAG). Then our framework is used to establish, (1) the communication system that allows these *actors* to send and receive data across each other and (2) the computation system that executes these tasks in parallel on a given multicore system. The main contributions can be described as,

1) A parallel multi-point communication layer, which provides the ability for stream actors to communicate across the network
2) An execution system that can run the given data-parallel filters on the multicore system
3) The utilization of network hardware features such as queues and flow control to support stream applications.

III. Introduction

Our main objective is to optimize performance of streaming applications by first, providing a specialized communication link between the filters and second, extracting data-parallel actors and executing them in parallel. Stream applications usually consists of stateless and stateful filters. Stateful filters are those that are depedendent on the previous execution which makes it difficult to replicate them. Stateless filters do not pose such limitations and can be freely replicated based on the available data streams. In this paper, we focus on extracting data parallelism based on stateless filters and streams. Example operations of such an application are represented in the stream graph shown in Figure 1. All the filters used in this application are essentially stateless. This application operates by performing encryption on the data that matches predefined rules like file names, access level of user, etc., otherwise it performs compression. Due to the nature of such stream applications, it is possible to replicate and distribute the filters across different CPUs. We are particularly interested in a scenario where groups of networked servers are used to perform these kind of stream operations by distributing data streams over multiple nodes.

The main challenge that arises in such a situation is how best to address the intensive computational and communication demands of such applications. Two important features enable us to tackle this challenge in our framework. Firstly, the operations of both the application and the network are

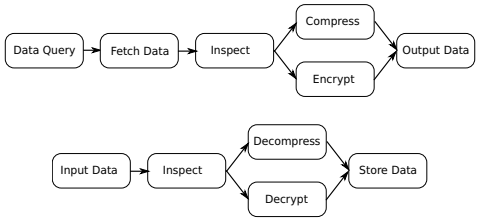

Figure 1: Sample stream application

designed to be stateless filters. Secondly, communication techniques are employed that enable multicore processors to perform computation and communication in parallel.

In building a framework for such applications on general purpose computers, we exploit two hardware features that could allow for dramatically improved performance, namely multicore processors and multi queue network controllers. Using multiple queues makes it possible to perform communication in parallel and by using the multiple cores, the filters from the stream application can be executed concurrently.

A major bottleneck for communication on general purpose computers is the overhead of the operating system's network stack. In particular, for applications that run as user processes, data must be copied between OS kernel space and user space. Previous research uses other mechanisms to access data from the network hardware directly to overcome this problem [11]–[13]. Most of these use modern network hardware in combination with specialized system software that allows direct access to hardware-controlled packet queues, dramatically reducing overheads and improving performance. However, constructing stream applications that use these features is difficult. Furthermore, many stream applications have a very specific communication pattern, typically operating on a *flow*, which is a specific *stream* between two filters. Data in a *stream* must be processed sequentially, whereas different *flows* can often be processed in parallel.

Newer network hardware has introduced mechanisms for separating arriving data into multiple queues, where data that belong to the same *flow* are placed in the same queue [17], [18]. We can utilize this feature to operate on *flows* by processing each of the queues in parallel. However, building applications that can use this feature directly is a challenging task because it requires explicit parallel programming and interaction with low-level network interfaces. Previous research [11], [12] that focuses on improving the performance of network applications in such a manner, shows the complexity involved in building such a system.

In our framework, we use a more elegant approach. We define a runtime framework that provides pre-built communication filters. In order to implement a stream application such as that shown in Figure 1, all we need to do is define the computation filters and connect them in the framework to the pre-built communication filters. On execution, our runtime framework is able to replicate the stateless filters and

the communication filters onto the different multiple cores, thereby executing the operations belonging to different *flows* in parallel. Since this paper describes the design of our framework, it concentrates on the low-level issues regarding network protocol, hardware and operating system calls and how we create high level filters to abstract these issues thereby making it easier to program while at the same time improving performance.

Section IV represents the design principles that we follow to conceptualize our framework. Section V describes the implementation details used in the construction of the system. In section V-B1, we discuss the challenges in using a system consisting of execution engines to manage the work. Finally in section VI-B, we present a simple stream application constructed from operations such as compression and encryption and evaluate our framework.

IV. Design Principles

The design of a high-level programming framework that integrates packet processing tasks into the application requires overcoming several challenges. In this section, we describe these challenges and the principles that are used to address them.

A. Communication filters

These filters represent simple operations responsible for transforming application data suitable for transmission over the network. We use a graph-based representation of filters to determine the dependencies involved for the different stages and to determine the relation with the different tasks of the application. This graph contains the stages through which the data associated with the application is converted into network packets and vice versa. It shows which of the tasks can be performed in parallel with those of the application, and highlights those that have a data dependency on a specific task in the application. The graph also enables optimizations based on specific characteristics of the application or the architecture used for execution. For example, Figure 2 shows a simple graph representing User Datagram Protocol (UDP) packet generation. As the data is passed from one filter to another, a particular operation transforms the data. For example the `Compute Checksum` filter handles the process of calculating the checksum of the data required by the UDP protocol and adds it to the location specified by `UDP Header` filter.

Figure 2: Simplified UDP packet construction

B. Userspace communication

For a communication system to be integrated with an application, it is necessary to have an efficient mechanism for reading and writing packets to and from the network interface controller (NIC) without interacting with the operating system. Rapid data access here is crucial for the extraction of data parallelism at later stages. We use a number of techniques to support this rapid data access.

1) Handling Multiple Hardware Queues: The development of virtualization and improved network flow handling has led to hardware-level multi queue support in NICs. During transmission, concurrent writes to different queues are possible, enabling multiple processes to send data simultaneously. During reception, the NIC classifies each packet onto one of the receive queues in a technique known as Receive Side Scaling (RSS) [17], [18]. This technique uses the header information or tuples to classify packets onto different hardware queues. Multi queue support can handle several streams of data in parallel and it is exploited extensively in our framework for this reason. By offloading the classification of packets to the NIC, we remove any overhead associated with packet classification in the application. We can also use these hardware queues as a means of balancing the workload across different threads by controlling the number of queues that each thread handles.

2) Flow based communication: A common technique known as `batching` is used to boost the performance of network applications, where several packets are combined and operated upon together, thus reducing the overhead of I/O operations. In our framework, batching is utilized in a form where the number of packets in a batch is varied based on the available space in the hardware queue. This number reflects the amount of data we are able to process before issuing hardware synchronization calls and forms an important means by which we reduce the usage of system calls.

3) Reduction of Per Packet System Calls: System calls are required in order to synchronize data between the NIC hardware and the buffer memory used for intermediate storage of data. Previous work describes the overhead associated with per-packet system calls and provides mechanisms for overcoming this overhead [12], [19], [20].

We employ an *adaptive scheme* described in section V-A that utilizes this strategy of reducing per-packet system calls. Our framework balances the system calls based on the communication rate and application processing speed. The transmission and reception are handled differently to optimize the usage of system calls in each of the cases independently. During transmission, we issue system calls to synchronize the available buffer space based on the rate at which we transmit data. This prevents degradation in latency for larger packets and improves performance for smaller packets. During reception of packets, the system reads packets at the rate at which it is able to process them.

C. Computation filters

In the stream processing model, the communication flow between tasks is well-defined. The computation filter con-

sists of operations that the user wishes to perform on the data. The design guideline is to split the application into as many filters as possible and let the runtime framework decide the scheduling and execution order, making it possible to extract more task parallelism. The filters associated with the computation are connected to the communication filters by specifying them in the form of a directed acyclic graph (DAG). The runtime system uses this graph to execute a particular filter while passing the corresponding data from one filter to another. The computation filters are designed by the user, and the runtime framework provides the necessary communication filters.

D. Integration

This stage connects the corresponding communication filter to the computation filter defined earlier. Based on the stream graph that is provided, the necessary communication filters are added. This also determines the unique *flows* that exists between the filters and is used later to replicate and execute them in parallel.

E. Parallel Processing Engine

Two important aspects that the parallel processing engine must address are the scheduling of filters and the balancing of the load across the available resources. The parallel processing engine consists of collaborating processes which are bound to specific CPUs and act as engines to execute the workload. Based on their requirements and the available resources, filters are scheduled across these execution engines. We assume a static schedule of the filters onto the given multiple cores is provided for execution.

V. Implementation

In commodity systems, the interaction with the OS for network operations is a necessary abstraction to handle multiple applications. However, in the case of specialized servers which cater to specific applications, it would be beneficial to interact directly with the network device. Moreover, in the case of intensive network traffic, the OS could act as a bottleneck due to the overhead associated with data flowing through the kernel before being transmitted through the network device [9], [10]. Eliminating this would be possible, either by executing the application in the kernel space (which could compromise stability of the system), or by using an interface for accessing the NIC buffers from user space. The latter approach is supported by several interfaces such as `netmap` [20], PF_RING DNA [19], etc., Even though these interfaces support user space packet access, they lack the ability to be invoked in a efficient manner by the application. More specifically, they do not have any way of optimizing the rate of processing, lacking, for example, the concept of flows for application tasks. These application tasks are represented as computation filters similar to those described in Figure 1. The lack of support for interacting with these interfaces, in order to read and write data in parallel from concurrent processes or threads, adds to the problem. Also, since the application is now responsible for handling network packets, it has to perform the additional operations that would otherwise be carried out by the OS.

Our framework provides support for sending and receiving data in parallel over the NIC from user space and a common parallel execution engine that supports both computation and communication filters.

A. Parallel communication interface

The parallel communication interface, when combined with the computation filters, provides concurrent communication for the application threads or processes over the network. To do this, two important requirements have to be met. Firstly, access to network data from user space should be provided, which can be read from and written to in parallel. Secondly, it should be possible to integrate this with the computation by providing the necessary communication filters that can perform packet operations. The ability to choose communication filters allows the flexibility in the choice of communication protocol.

1) Userspace data access: Several researchers [19], [20], etc., propose access to network buffers from user space. We use `netmap` APIs [20] to access data from the network buffers. `Netmap` uses a set of ring buffers in kernel space and user space to export the network hardware buffer's memory region. This enables applications in user space to directly access the network hardware.

We leverage the availability of multiple hardware queues present in modern NICs to read and write data concurrently. `Netmap`, by default, supports assigning each hardware queue to an individual application process or thread. However, in order to allow for more efficient communication in the framework, we require the ability to assign more than one queue to a particular thread or process. Our modifications to `netmap`'s hardware queue assignment API allows our framework to balance the amount of data handled by each process by varying the number of queues assigned to it.

During testing, it was found that it was essential to have control over the rate at which system calls were issued with respect to the amount of data being transferred. Frequent system calls hamper bandwidth and limit the peak performance of the application. To tackle this, we propose the following algorithms for reading and writing data from the hardware queues using the `Netmap` APIs. These algorithms are designed to provide the following features,

- Provide batching of packets based on available space on hardware rings
- Reduce the usage of system calls such as *ioctl* and *poll*
- Provide mechanisms for handling multiple queues through the application.

Algorithms 1 and 2 use a batch update for sending and receiving packets based on the rate at which data flows through the interface. This is done by maintaining a data structure that has information about the hardware rings, updating it on a periodic basis. In cases where the application is generating sparse traffic, it is possible to frequently update the ring in order to maintain a consistent latency in transmission. Choosing a specific queue to send or receive data is determined based on an application specific function. The $f(n)$ present in Algorithm 1 line 9 and Algorithm 2 line 9 represents this. Choosing a random queue ($f(n) = X \sim U([0, n])$ n = no of queues) has proven to be effective in our experiments, but if an application requires a specific flow to be maintained, a different function here could provide that support. The $threshold$ specified in Algorithm 2 line 18 is determined for each application based on the rate at which it is able to process data. By issuing the system calls such as *ioctl* and *poll* only when required, and effectively handling the multiple queues, we are able to achieve improved performance.

Algorithm 1 To send packets

```
1: struct ring[n]{
2:                 ▷ Stores information about hardware rings
3:                         ▷ n ← tx rings assigned to thread
4:   avb, available space
5:   curr, current position
6:   limit, total space  }
7: function SEND_DATA_MQUEUE(Ring f(n), Data packet)
8:     while !success do
9:         i ← f(n)
10:        success = SEND_DATA(i, packet)
11:    end while
12: end function
13: function SEND_DATA(Ring i, Data packet)
14:    if ring[i].avb > 0 then
15:        hw_ring[i].slot(ring[i].curr) ← packet
16:        ring[n].avb ← ring[i].avb − 1
17:        ring[i].curr + +
18:        if ring[i].avb == 0 then
19:            hw_ring[i].avb − = ring[i].limit
20:            hw_ring[i].curr ← ring[i].curr
21:        end if
22:        return success
23:    end if
24:    issue ioctl and poll
25:    ring[i].avb ← hw_ring[i].avb
26:    ring[i].limit ← hw_ring[i].avb
27: end function
```

2) Packet processing operations: Packet processing operations consist of converting the raw data that the application provides into network packets suitable for transmission. Several user space libraries exist that perform packet related processing operations like libnids [21], lwip [22], Click [13] etc. Since our application is targeted at a closed environment, we chose the Click modular router due to its flexibility and vast library of elements. Our current framework supports the UDP protocol but Click's modular infrastructure makes it easy to support more complex ones.

Algorithm 2 To receive packets

```
1: struct ring[n]{
2:                 ▷ Stores information about hardware rings
3:                         ▷ n ← rx rings assigned to thread
4:   avb, available packets
5:   curr, current position
6:   limit, total packets  }
7: function RECEIVE_DATA_MQUEUE(Ring f(n))
8:     while !packet do
9:         i ← f(n)
10:        packet = RECEIVE_DATA(Ring i)
11:    end while
12: end function
13: function RECEIVE_DATA(Ring i)
14:    if ring[i].avb > 0 then
15:        packet ← hw_ring[i].slot(ring[i].curr)
16:        ring[i].avb ← ring[i].avb − 1
17:        ring[i].curr + +
18:        if ring[i].avb < threshold then
19:            hw_ring[i].avb− = ring[i].(limit − avb)
20:            hw_ring[i].curr ← ring[i].curr
21:            ring[i].avb ← hw_ring[i].avb
22:            ring[i].limit ← hw_ring[i].avb
23:        end if
24:        return packet
25:    end if
26:    issue ioctl and poll
27:    ring[i].avb ← hw_ring[i].avb
28:    ring[i].limit ← hw_ring[i].avb
29: end function
```

B. Parallel Processing Engine

After defining the communication filters by means of `Click`'s infrastructure and access to network hardware through `netmap`'s APIs, we integrate these filters with the application tasks. The result is a complete stream graph that does both the computation and communication operations. The next logical step is executing these filters in parallel by exploiting data parallelism. Packet streams are inherently operable in a data parallel fashion i.e. it is possible to operate on two different streams simultaneously.

1) Execution engine: The execution engine consists of a set of processes and threads that the work is scheduled

upon in order to be executed. When implementing threads to execute code which consist of system calls such as *malloc(), ioctl(), poll(),* etc., we found a performance degradation, even when the threads are completely independent. To support the lightweight style of threads, the OS implements most of the locking mechanisms for system calls from threads. Previous research such as [23]–[25] shows the degradation in performance due to the usage of *malloc()* in threads. Since our framework has to deal with system calls and issue them in parallel, we use *unix process* to perform the tasks concurrently. We do this by using Message Passing Interface (MPI) [26] to construct a set of processes that is scheduled onto the different CPUs on a multicore system.

In order to prevent unwanted process migration due to the OS scheduler, we specify *affinity* of the MPI processes to particular CPU cores acting as execution engines on which different operations are scheduled. Previous research [27], [28] highlights the importance of affinity in network sensitive applications. Next, by utilizing our *parallel communication interface* (V-A), we specify the hardware queues each of these processes handle.

2) Parallel processing of computation filters: Once we have scheduled our execution engines on the physical system, the next step is to execute the stream graph for different *flows*. To extract data parallelism, we replicate the stream graph onto multiple execution engines. This ensures operations on the same *flow* are performed on the same CPU, thereby using local cache. By making sure that data belonging to a specific *flow* always ends up in the same execution engine, we only perform operations conforming to that stream of data.

VI. Experimental Evaluation

In order to evaluate our framework, we test the scalability and the performance speedup on a multicore system for an application that is constructed with the framework. Using two generic operations: compression and encryption, we build simple stream applications. We then measure the performance improvement as we increase the number of execution engines dedicated to the application. Although the framework can be used with a greater variety of more complex benchmarks, these simple benchmarks allow us to evaluate the framework's performance without unnecessary application-specific complications.

A. Hardware Setup

The evaluation system consists of two servers connected directly to each other. Each server consists of two Intel Xeon 5600 hexacore HT in a dual socket configuration with 24GB DDR3 memory. In total there are 2 Processors x 6 Cores x 2 HT = 24 CPUs in each server. These servers are connected using Intel 8299EB 10Gbe adapters over directly attached twin-axial cables. The adapters are located in the PCI Express x8 slot to provide required bandwidth for the network cards. This forms a closed connection between two servers and limits the influence of external devices such as switches or routers on data transmission.

Each application is constructed in two parts, one a source of data and the other a sink. We measure the overall time it takes for application task operations, communication operations, and the time to send from the source and receive at the sink. This is done for different packet sizes and applications. Then we plot the speedup achieved in each of the cases over the serial version, where we have a single sender and receiver operating on the multiple queues. This is the baseline in our experiments and is similar to how the application would behave if it were implemented without our framework but still using the system software to directly access the network hardware.

The Intel 82599EB is restricted to 16 queues when using Receive Side Scaling (RSS). This limited the maximum amount of processes at the sink to be 16, even though 24 CPUs were available. The source is not affected by this limitation and can use all the 24 CPUs available, along with 24 hardware transmission queues. Overall, we require at least one hardware queue mapped to each of our processes or execution engines.

We only use two nodes connected directly with each other as the experimental setup. This establishes an ideal setup to empirically evaluate the performance gain obtained due to our framework.

B. Application Construction

In order to evaluate our framework, distinct operations were required that would be suitable for a streaming framework. Since compression and encryption applications are relatively common, we chose two applications composed from these operations. The stream graph that represents both these applications is shown in Figure 3. The UDP protocol is used for transmission to send the data between source and sink.

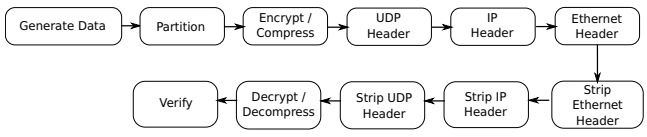

Figure 3: Integrated Task Graph

Both applications are constructed similarly. Data sizes of 1.28GB and 2.56GB are selected as the workload for the compression and encryption applications respectively. The OpenSSL library is used for encryption/decryption and zlib is used for compression/decompression operations. Each application consists of a data generation phase, then data is compressed or encrypted depending on the application over the two nodes. These are representative of some of the typical operations that would be involved in a much more complex application. Use of these simpler stream applications enables us to evaluate the performance of our

framework and not that of any optimizations specific to the application.

MPI's communication calls are at a higher level and rely on the Linux kernel for handling network operations, unless there are specialized interconnects such as Infiniband which provide dedicated hardware for these operations. As a result, a corresponding MPI implementation would suffer from the substantial overheads resulting from frequent kernel calls. So in our comparison we chose to measure the speedup we achieve only due to our communication system when utilized by different parallel processes. As regards to scalability, an improperly configured network topology or poor load balancing can adversely affect an MPI implementation in the same way that it could affect our system.

VII. Results

It can be seen in Figure 4 that there is considerable difference in the way each of the operations scale. Three distinct observations can be made from the results.

One is that, from Figure 4.(d,e,f), we note that compression requires more work than decompression, whereas from Figure 4.(a,b,c) encryption and decryption are more balanced. This is evident from the continued speedup obtained from up to 4 senders in the case of compression/decompression. Only a slight asymmetry can be seen in the encryption/decryption case, where adding more senders than receivers only yields a slight increase in performance. In this case, the likely explanation is support for AES instructions in the architecture, making decryption a marginally easier task. In general though, the performance here only improves by increasing both the number of senders and receivers simultaneously. Even from these preliminary results, it is clear that the optimal sender/receiver balance varies from application to application and highlights the need for effective load-balancing.

Packet size also impacts performance as seen in Figure 4.(d,e,f). The compression operation is influenced more than that of the encryption application. There is a speedup of 12x in Figure 4.(d) with smaller packets and 10x in Figure 4.(f) with larger packets. This discrepancy appears to be greater in cases where there is a large number of senders and a high number of receivers. A possible explanation for this relates to the CPU utilization on the faster receivers. With large packet sizes, they are more likely to block, waiting for data to be streamed from the slower senders. Smaller packets allow the work to be distributed more evenly among the available receivers. In the encryption application, there is not as much asymmetry and packet size does not matter as much. The fact that packet size impacts applications differently suggests that it is essential to incorporate dynamic packet-size selection into the framework itself.

From Figure 4(a,b,c,d,e,f), a drop in speedup is apparent when employing 12 senders or more. The most likely explanation for this is the influence of the NUMA architecture [29]. In such a architecture there are two or more physical CPUs connected with each other using a high bandwidth bus. Due to our hardware setup when we use more than 12 parallel processes we start to utilize the second CPU. Due to the limitation in the way our data is processed, all the nodes try to access the memory belonging to a single CPU thereby causing cache misses and leading to a drop in performance. Eventually, since the processing requirement is more than this bottleneck we see a steady increase in performance. Despite these NUMA architecture issues, once the initial drop is overcome, the application continues to scale and eventually attain better performance in most cases.

In summary, it can be seen that the framework scales well for different workloads, can handle both computations and communication, and can operate on them in parallel. We see that we can achieve a speedup $10\times$ for compression and $12\times$ for encryption based stream applications respectively on an architecture consisting of two nodes of 24 logical cores each. In some cases, the framework can be improved by having a greater degree of control over the fine grained parallelism within the execution engines. This can be achieved by introducing pipeline stages for different operations.

Since we work with a static schedule provided by the user in executing the applications, dynamic load balancing is likely to be a beneficial addition to the framework, in terms of performance. It is also possible to add the capability to dynamically adjust the number of network queues allocated to each core as a stream application's requirements change over time. In the future we would also like to extend our framework to work with larger number of nodes. Finally, as seen in Figure 4, being able to adjust packet sizes dynamically to an optimal value for each network link, dependent on the stream application, would also be a useful addition to the framework.

VIII. Related Work

Stream programming languages such as StreamIt [1] and LUSTRE [2] target stream applications, but the focus is more on parallelizing the computations related to the application alone on multicore systems, unlike our runtime framework which also parallelizes communication operations.

The Stream Project [30] concentrates on researching multiple facets of middleware for a large scale system with an emphasis on problems such as data storage, redundancy, and real time systems. In comparison, our work is more about providing a simple system using which we can improve performance of stream applications by providing filters that can be used to interact with the hardware without the usage of the traditional Linux subsystem. Ultimately, our work could feed into such large scale middleware systems to improve communication and computation performance.

MPICH-G2 [31] targets large grid networks and applications written using the MPI programming model, unlike our

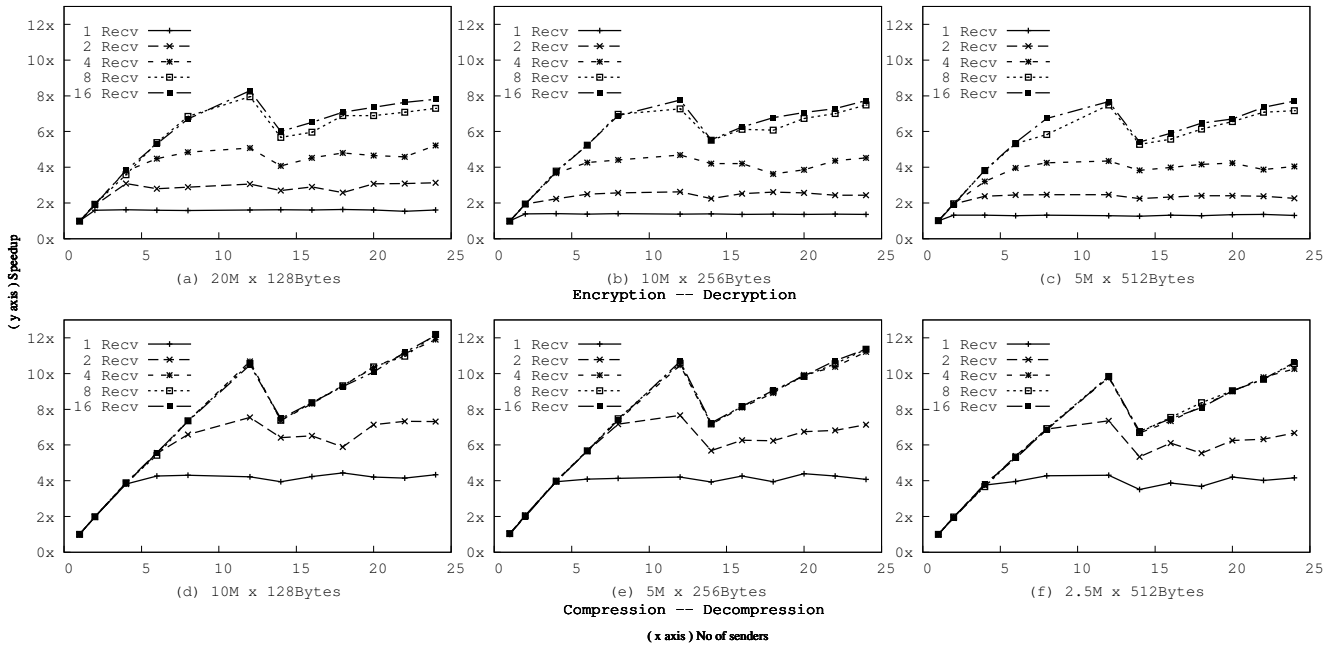

Figure 4: Performance Speedup; (a),(b),(c) - Application A. Encryption; (d),(e),(f) - Application B. Compression; Speedup based on time in takes to compute a fixed workload for different number of source and sink processes and for varying packet sizes over a single source and sink are shown above

focus on the streaming model. Moreover, we do not employ any of the MPI communication primitives provided to handle streaming data. Instead we use our own fast runtime system to communicate data between the filters in the stream graph across systems.

Wagner et al. [32] propose a stream processing framework that completely utilizes the MPI library. They make use of MPI groups and communicators to improve the flexibility of the MPI library to support stream processing. Their approach employs a library using MPI primitives to construct their stream processing system and relies on the underlying abstraction layer to communicate over any network the MPI library supports. Our runtime framework is designed to provide a low overhead, parallel communication framework by directly utilizing the network link.

Mancini et al. [33] propose a hybrid approach of Stream and MPI programming models, and use a whole MPI program as a building block in a stream application to improve processing speeds of computational units in hetrogeneous computational systems.

While we use MPI to execute the computation filters, our runtime framework employs a direct and parallel communication interface for the actual transmission and reception of data between filters. This is the key feature that enables us to parallelize the communication along with the computation for a given streaming application, which is essential in improving its performance.

IX. Conclusions

In this paper, we have proposed a parallel runtime framework that can integrate communication and computational operations in stream applications and perform both types of operation in parallel.

Our algorithms for sending and receiving data implicitly perform batching and reduce the usage of system calls, and we combine this with `netmap` APIs to minimize the interaction with the OS. This allows communication performance in our framework to better scale with the number of available processors. Instead of performing the communication operations separately in the OS, by integrating them within the stream application, we are able to construct a unified stream graph that represents the entire application, including computation and communication filters. Using this, we are able to parallelize stream applications and achieve speedups of more than a factor of eight in all the applications we tested.

By integrating these features into a runtime framework, the system of multiple queues, parallel communication, and computation are entirely hidden from the programmer, who merely specifies a standard stream graph for computation alone. This frees the programmer from the onerous task of dealing with the implementation of the substantial communication requirements that result from the distribution of stream applications across multiple systems.

The results show that our system scales to as many parallel processes as there are CPUs on our 24 multicore system, and achieves speedups of more than a factor of ten in some cases compared to sequential implementations.

Acknowledgment

This work is funded by the IRCSET Enterprise Partnership Scheme in collaboration with IBM Research, Ireland.

REFERENCES

[1] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "StreamIt: A Language for Streaming Applications," in *Proceedings of the 11th International Conference on Compiler Construction*, ser. CC '02. London, UK, UK: Springer-Verlag, 2002, pp. 179–196.

[2] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous dataflow programming language LUSTRE," in *Proceedings of the IEEE*, 1991, pp. 1305–1320.

[3] R. Stephens, "A survey of stream processing," *Acta Informatica*, vol. 34, no. 7, pp. 491–541, 1997.

[4] S. Muthukrishnan, *Data streams: Algorithms and applications*. Now Publishers Inc, 2005.

[5] V. Agarwal, D. A. Bader, L. Dan, L. Liu, D. Pasetto, M. Perrone, and F. Petrini, "Faster fast: multicore acceleration of streaming financial data," *Computer Science-Research and Development*, vol. 23, no. 3, pp. 249–257, 2009.

[6] L. Golab and M. T. Ozsu, "Data Stream Management Issues A Survey," Tech. Rep., 2003.

[7] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss, "Quicksand: Quick summary and analysis of network data," AT&T Labs-Research, Tech. Rep., 2001.

[8] M. Sullivan and A. Heybey, "Tribeca: a system for managing large databases of network traffic," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ser. ATEC '98. Berkeley, CA, USA: USENIX Association, 1998, pp. 2–2.

[9] W. Wu and M. Crawford, "Potential performance bottleneck in Linux TCP," *Int. J. Commun. Syst.*, vol. 20, no. 11, pp. 1263–1283, Nov. 2007.

[10] W. Wu, M. Crawford, and M. Bowden, "The performance analysis of linux networking - Packet receiving," *Comput. Commun.*, vol. 30, no. 5, pp. 1044–1057, Mar. 2007.

[11] M. Dobrescu, N. Egi, K. Argyraki, B. gon Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "RouteBricks: Exploiting Parallelism to Scale Software Routers," in *In Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[12] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: A GPU - accelerated software router," *SIGCOMM Comput. Commun. Rev.*, vol. 40, pp. 195–206, Aug. 2010.

[13] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click Modular Router," *ACM Trans. Comput. Syst.*, vol. 18, pp. 263–297, August 2000.

[14] O. B. Fredj, H. Sallay, M. Rouached, A. Ammar, A.-S. Khaled, and M. B. Saad, "Survey on Architectures and Communication Libraries dedicated for High Speed Networks," *Journal of Ubiquitous Systems & Pervasive Networks*, vol. 3, no. 2, pp. 79–86, 2011.

[15] A. Romanow and S. Bailey, "An Overview of RDMA over IP," in *In First International Workshop on Protocols for Fast Long-Distance Networks (PFLDnet)*, 2003.

[16] W. Feng, J. Hurwitz, H. Newman, S. Ravot, R. L. Cottrell, O. Martin, F. Coccetti, C. Jin, X. Wei, and S. Low, "Optimizing 10-Gigabit Ethernet for Networks of Workstations, Clusters, and Grids: A Case Study," in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, ser. SC '03. New York, NY, USA: ACM, 2003, p. 50.

[17] Microsoft Corporation, Receive-Side Scaling Enhancements in Windows Server 2008, website:http://msdn.microsoft.com/en-us/library/windows/hardware/gg463253.aspx accessed:May 2013.

[18] Intel Corporation, Intel 82599 10gbe Controller Datasheet website:http://www.intel.com/content/www/us/en/ethernet-controllers/82599-10-gbe-controller-datasheet.html accessed:May 2013.

[19] L. Deri, "nCap: wire-speed packet capture and transmission," in *Workshop on End-to-End Monitoring Techniques and Services*, may 2005, pp. 47 – 55.

[20] L. Rizzo, "Revisiting Network I/O APIs: The netmap Framework," *Queue*, vol. 10, no. 1, pp. 30:30–30:39, Jan. 2012.

[21] Libnids website: http://libnids.sourceforge.net accessed: May 2013.

[22] A. Dunkels, "Design and Implementation of the lwIP TCP/IP Stack," *Swedish Institute of Computer Science*, vol. 2, p. 77, 2001.

[23] C. Lever and D. Boreham, "malloc() performance in a multithreaded linux environment," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ser. ATEC '00. Berkeley, CA, USA: USENIX Association, 2000, pp. 56–56.

[24] D. Dice and A. Garthwaite, "Mostly lock-free malloc," *SIGPLAN Not.*, vol. 38, no. 2 supplement, pp. 163–174, Jun. 2002.

[25] M. M. Michael, "Scalable lock-free dynamic memory allocation," *SIGPLAN Not.*, vol. 39, no. 6, pp. 35–46, Jun. 2004.

[26] The MPI Forum, "MPI: A message passing interface," in *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, ser. Supercomputing '93. New York, NY, USA: ACM, 1993, pp. 878–883.

[27] J. D. Salehi, J. F. Kurose, and D. Towsley, "The effectiveness of affinity-based scheduling in multiprocessor network protocol processing (extended version)," *IEEE/ACM Trans. Netw.*, vol. 4, no. 4, pp. 516–530, Aug. 1996.

[28] A. Foong, J. Fung, and D. Newell, "An in-depth analysis of the impact of processor affinity on network performance," in *Networks, 2004. (ICON 2004). Proceedings. 12th IEEE International Conference on*, vol. 1, Nov. 2004, pp. 244 – 250 vol.1.

[29] M. Awasthi, D. W. Nellans, K. Sudan, R. Balasubramonian, and A. Davis, "Handling the problems and opportunities posed by multiple on-chip memory controllers," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 319–330.

[30] Stream Project website: http://www.streamproject.eu/ accessed:May 2013.

[31] N. T. Karonis, B. Toonen, and I. Foster, "MPICH-G2: a Grid-enabled implementation of the Message Passing Interface," *Journal of Parallel and Distributed Computing*, vol. 63, no. 5, pp. 551–563, 2003.

[32] A. Wagner and C. Rostoker, "A lightweight stream-processing library using MPI," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, May, pp. 1–8.

[33] E. Mancini, G. Marsh, and D. Panda, "An MPI-Stream Hybrid Programming Model for Computational Clusters," in *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, May, pp. 323–330.