A high-level programming framework to support parallel processing of communication intensive stream applications

Servesh Muralidharan
School of Computer Science and Statistics
Trinity College Dublin
Dublin 2, Ireland
Email: muralis@tcd.ie

David Gregg
School of Computer Science and Statistics
Trinity College Dublin
Dublin 2, Ireland
Email: David.Gregg@cs.tcd.ie

***Abstract*—Stream applications are often limited in their performance by the communication system employed. In systems that use conventional 1GbE or 10GbE standard, the Operating System (OS) handles most if not all of the network operations. In such cases, the communication stack, which was not designed to handle such tremendous amounts of data, acts as a bottleneck and restricts the performance of the application. An optimum framework for computing and communicating in parallel is essential for the continued improvement in performance.**

We propose a high-level programming framework that integrates the packet processing operations with the stream applications and provides a common interface to execute both the application and packet processing operations in parallel on the multicore processors.

We utilize the ability of network interface controller (NIC) to classify packets on to multiple hardware queues. We combine this with system software that allows direct access to the hardware queues. We develop an interface in our framework for application processes to read or write to in parallel. Our system integrates this with the application by a series of packet processing operations that transform the data from the application to packets suitable for transmission. Representing these tasks as stages in the application itself, we are able to parallelize them using techniques similar to those used for parallelizing regular computations.

All the operations required for the communication are performed in the application, bypassing the OS to send and receive data directly over the network interface. By overcoming the bottleneck induced by performing network operations in the OS and by parallelizing both the computations and packet operations together, we are able to boost the performance of the application.

I. Motivation

Stream applications are designed in a manner in which they operate on data that flows as a *stream*. This data stream passes through different stages of computations, to be transformed from query to result [1] [2] [3]. This is different from conventional computation model where data is stored and then processed. By eliminating the need to store data and processing it on the go, it makes it possible for stream applications to compute on large amounts of data.

Stream applications such as file storage systems, database query management, data mining, etc., that operate on vast amount of data are often distributed over multiple systems. Traditional stream applications do target multicore processors for performing their computations in parallel. Due to this and the exponential rate at which multicore processors are growing the amount of computations performed by these stream applications would further increase and eventually lead to processing even larger amounts of data. With the growth in the complexity of these applications it is necessary to leverage on a multicore environment to improve their communication performance as well. Such stream applications that operate on large amounts of data at very high bandwidth are often limited in their performance by the communication stack of the OS kernel [4] [5] [6] [7] [8]. In the continued race for improvement of multicore and many core architectures, it is becoming essential to improve the processing speed of packet operations along with those of the application in order to scale the performance of the application.

When performing packet processing operations for communication intensive stream applications, a major challenge is the very high levels of performance required. These applications process large numbers of data streams in real time. If the communication system cannot keep up with the rate at which packets arrive due to these data streams, then they must be dropped or remain unprocessed.

In building a framework for such applications on general purpose computers, it may be possible to exploit two hardware features which are multicore processors and multi queue network controller that could allow dramatically improved performance. Using this it makes it possible to do much of the packet processing in parallel using the multiple cores that are available on modern processors.

A major bottleneck for packet processing 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 use alternate mechanism to access packets from the network hardware directly due to this [6] [7] [8]. Most of these use modern network hardware in combination with a particular systems software that allows

direct access to hardware-controlled packet queues to dramatically reduce overheads and improve performance. However, constructing stream applications that use these features is difficult. Furthermore, many stream applications have a very specific communication pattern, typically operating on data between a source and sink in the form a *data stream*. Data in a stream must be processed sequentially, whereas those from separate streams can often be processed in parallel.

Modern network hardware typically has a mechanism for separating arriving packets into multiple queues, where packets that belong to the same flow are guaranteed to be placed in the same queue [9] [10]. In many stream applications we can utilize this to operate on multiple streams of data by processing each of the queues in parallel. However, building such applications is complex because it requires explicit parallel programming, and it requires explicit interaction with the low-level network interfaces. Research focuses on improving the performance of network applications by parallelizing the packet processing tasks on multicore architectures [6] [7]. We parallelize both the application and packet operations to gain significant improvement in performance.

II. Contribution

In this paper we propose a high-level programming framework for building communication intensive stream applications. We focus on integrating the packet processing and application tasks into a single graph and how we use this to extract data parallelism. The developer describes their program as a graph of *actors*. Each actor consumes one or more *tokens* of data, which may be simple data such as integers, or more complex data such as packets, arrays or objects and produces one or more tokens. The framework can be used to automatically map the application over multiple parallel cores, and interface with the `netmap` APIs [11] for accessing data through the network buffers.

The system can be divided into three main parts:

- A subsystem for interfacing with the network hardware through the `netmap` APIs to send and receive data from the application in user-space without going through the operating system. (Section V-A)
- Given a graph that represents the workload to be performed by the application, we construct and integrate the packet processing operations represented as a directed acyclic graph (DAG) to it. (Section V-B1)
- A run-time system that executes these tasks in parallel and explicitly schedules them on the multicore processor. (Section V-B2)

Even though we integrate the system with the application the techniques for extracting parallelism from the application is not in scope of this paper. The application is considered to be in a streaming model and the tasks corresponding to the various functions are considered to be parallel in nature.

III. Background

Let us consider a sample application that operates on data streams. The important operations of the application are represented in the stream graph shown in figure 1. The application operates by performing encryption on the data that matches predefined rules like file names, access level of user, etc., otherwise it performs compression and store the information in the location specified. Considering a scenario in which, a group of servers are used to perform these operations by evenly distributing them over the different nodes.

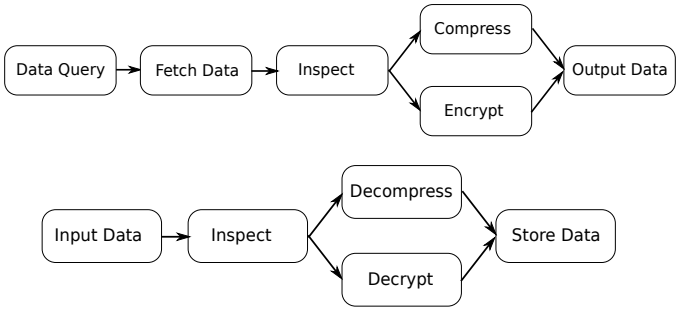

Fig. 1: Sample stream application

The main challenge that arises in such a situation is how to tackle such large number of computations from the application and the network operations due to the intensive communication. Two important features enable us to tackle this challenge posed by such a system. Firstly, the operations of both the application and the network is parallel in nature and secondly, the presence of multicore processors to execute the application.

In our system we use an efficient parallel processing framework that could leverage on multicore processors and multi queue network controllers to achieve this. The performance of multicore processors has dramatically improved the capability of commodity hardware. With this we would be able to meet the computational demand of the application. In addition, the presence of multi queue network interface controller which was designed to handle multiple streams of data could help us achieve the necessary communication performance.

An important part of our system is the tasks related to packets such as Userdatagram Protocol (UDP), Internet Protocol (IP), etc., We utilize the click modular router for these packet processing tasks. Click modular router is a tool that includes a library of elements which is used in building flexible, re-configurable and modular software routers [8]. It offers a platform that is flexible for rapid development of newer network protocols and achieves good performance. An improved version known as SMP Click supports thread based parallel processing of network related tasks [12]. Until recently click's performance in the user space was not as good as its kernel space equivalent. This is mainly due to its extensive use of polling driver in the kernel space that was not possible in user space. Netmap I/O [11] provides a set of APIs that could directly access the packets from the NIC and it helped in the recent improvement of click's user space performance. We found the simpler construction of `netmap` easier to adapt to our framework compared to other user space packet access APIs [11] [13] [14].

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 show the challenges involved in using a system consisting of processes that are used to actually perform the work. The extent to which we extract parallelism ranges not only to the packet tasks but to the low level access of data I/O to the network interface hardware through certain additions to the `netmap` APIs. Finally in Section VI-B we present a simple application constructed from operations such as compression and encryption in order to evaluate our high-level 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 tackle them.

A. Userspace Packet I/O

For a packet processing system to be integrated with an application it is necessary to have an efficient mechanism for accessing the data from the NIC without interacting with the operating system. Rapid data access here is crucial for the extraction of data parallelism at later stages.

1) Handling Multiple Hardware Queues: The development of virtualization and improved network flow handling has lead to hardware level multi queue support in the NICs. During transmission concurrent writes to different queues are possible, enabling multiple processes to send data simultaneously. During reception the NIC classifies the packet on to one of the receive queues using a technique known as Receive Side Scaling (RSS) [9] [10]. It uses the header information or tuples to classify packets on to different hardware queues. Multi queue support can handle several streams of data in parallel and it is exploited extensively in our framework to process data concurrently. By offloading the classification of packets to the NIC, it removes any overhead associated with packet classification in the application. We use these hardware queues as a means of not only accessing the data in parallel but balancing the workload across different threads. This is done by controlling the number of queues that each process handles based on the rate at which packets are received in each of the queues. It is necessary to have different processes accessing the transmit and receive queues to maximize concurrent access to data.

2) Flow based Packet Batching: Packet batching is a common method used to boost the performance of network applications. Packet batching combines several packets and performing operations on them together, thus reducing the overhead involved in the I/O operations.

In our framework we utilize batching but the number of packets that forms a batch is not fixed but varies 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 a 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 packets. Previous work describe the overhead associated with per packet system calls and provide means of optimizing this [13] [7] [11].

We employ an *adaptive scheme* described in section V-A that uses the above principle. 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 case 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 given the same bandwidth at which we transmit in both the cases. During reception of packets the system reads packets at the rate at which it is able to process them.

B. Packet Processing Graph

We use a graph based representation of packet processing tasks to determine the dependency involved for the different stages and determine the relation with that of the different tasks of the application. The graph contains the stages through which the data, associated with the application, is transformed 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 data dependency on a specific task in the application. This graph also enables optimizations based on the characteristics of the application or the architecture used for execution. Figure 2 shows a User Datagram protocol (UDP) packet generation represented in this graph.

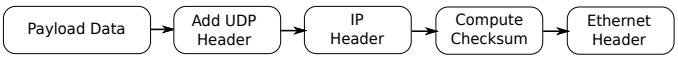

Fig. 2: Simplified udp packet construction

C. Application Task Graph

The subset of the application that we are interested in parallelizing is the stream processing component. In such components the communication between tasks is predictable and there is correlation between the stages of the application. Both the packet processing and application tasks can be represent by a directed acyclic graph (DAG) similar to the packet processing graph. The packet processing tasks operate on packets and the application tasks operate on the data. Since we focus more on parallel packet processing, the optimization in relation to the application graph are not discussed. It is assumed that a static version of the application graph is available and we focus more on executing these tasks in parallel with that of the packet processing tasks.

D. Integration

The sequence of tasks that can be executed in parallel is determined at this stage. The integration stage is used to combine the application graph and the packet processing graph into a larger graph that unifies and represents the overall operations. The dependency relations determined on the previous cases are used in establishing links between the graphs. We determine the sequence of tasks that can be executed in parallel from both the application task graph and packet processing graph. The combined set of tasks is then used by the parallel processing engine.

E. Parallel Processing Engine

Two important aspects that the parallel processing engine must tackle is the scheduling of tasks and balancing the load across the available resources. The parallel processing engine consists of processes which are bound to specific core and act as engines to execute the workload. It is constructed in a manner in which its flexible and can adapt to utilize the resources efficiently. The tasks are scheduled across these execution engines based on the available resources and its requirements. In this manner the workload is distributed uniformly while retaining control over the different execution elements.

V. Implementation

Most of the existing stream applications depend on the operating system for most of their communication tasks. In commodity systems, the interaction with the operating system 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 operating system could act as a bottleneck due to the overhead associated with data flowing through the kernel before being transmitted through the network device [4] [5]. Eliminating this would be possible by either 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 the user space itself. The later case of accessing the buffers of network hardware in user space is supported by several interfaces such as `netmap` [11], PF_RING DNA [13], UIO-IXGBE [14], etc., Even though the exported APIs support user space packet access they lack the ability to be invoked in a efficient manner by the application. The lack of support for interacting with these APIs in order to read and write data in parallel from the different process or threads adds to this problem. Moreover since the application is responsible for handling network packets it has to perform the additional operations that would otherwise be carried out by the operating system.

The framework provides support for sending and receiving data in parallel over the network interface from user space and a common parallel execution engine that supports both the application and packet processing tasks.

A. Parallel Packet I/O

The parallel packet I/O interface when combined with the application tasks provides concurrent communication by the application threads or processes over the network. To do this two important requirements are to be met. Firstly, it should provice access to network data from user space which we can write to in parallel and secondly, to integrate it with the application by providing necessary interfaces for performing the packet operations.

1) Userspace Packet Access: Applications executed in the kernel space for accessing hardware devices directly would lead to instability of the entire system. Instead, several previous work [11], [13], [14], etc., propose access of network buffers from user space. We use `netmap` APIs [11] to access data from the network buffers. `Netmap` uses a set of ring buffers in the kernel space and user space to export the network hardware buffer's memory region. This enables the applications to execute in the user space and directly access the network hardware. It prevents the application from accessing memory outside the valid region by checking limits of the ring buffer. Packets can be written to the buffers when the space is available and an *ioctl* call to the driver would enable the network controller to send the data. A *poll* interface is implemented on the memory region mapped in the user space to check when new packets are available. On reading or writing the packets the available space is updated.

We leverage on the availability of multiple hardware queues present in modern NIC to read and write data concurrently to the hardware. `Netmap` by default supports assigning each hardware queue to an individual application process or thread, but we noticed that in order to be efficient in handling the interface we had to assign more than one queue to a particular thread or process. We modified `netmap`'s hardware queue assignment API to support this. Later, it also allowed the framework to balance the amount of packets handled by each process by varying the number of queues assigned to each of it in order to balance the workload.

We also found that it was essential to have a balanced control over issuing the system calls with respect to the amount of data being transferred. Frequent system calls seem to hamper the bandwidth and latency and in essence affect 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. The algorithms are designed to provide these features,

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

By minimizing the synchronizations with the hardware rings we essentially limit the system calls issued and improve the overall system performance.

Algorithm 1 & 2 uses a batch update for sending and receiving packets based on the rate at which data flows through

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
```

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
```

the interface. This is done by maintaining a data structure that has the information about the hardware rings and updating it on a periodic basis. Once a fixed amount of space is determined we update this on our local structure, we then continue to read or write data from the hardware ring but update the actual limits and position on our local structure. When we have used up the space that was allocated we then update the hardware ring with the changes done. In cases where the application is generating sparse traffic it would be possible to frequently update the ring in order to maintain a consistent latency in transmission. Multi queue support is provided by managing the process of selecting the ring to send or receive packets based on a application determined function. The $f(n)$ present in Algorithm 1 line 9 and Algorithm 2 line 9 represents this. Choosing a random queue usually proves to be effective in our experiments but if the application requires a specific flow to be maintained a different function here could provide that support. Cases where this would be helpful is if the data operations are more predictable or if the transactions seem to follow a specific pattern. 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.

2) Packet Processing Operations: Packet processing operations consists of converting the raw data that the application provides into network packets suitable for transmission. Since the application tasks are represented as a set of operations on a stream of data we include these operations as stages through which the data flows through to get transformed into data packets. By integrating them with the application task graph it makes it easier to extract parallelism later in the execution engine. Several user space libraries exist that perform packet related processing operations like libnids [15], lwip [16], click [8] etc., Since our application are targeted at a closed environment, we chose the click modular router for its flexibility and vast library of elements as a means of operating on the network packets. Our current framework supports a simpler udp protocol but click's modular infrastructure would make it easier to construct more complex ones.

Each of the element in the click contains a virtual function which is called from the previous element it is attached to. We had to overcome this limitation of using click's infrastructure to construct based on dynamically initialized objects and virtual functions. This was necessary due to two important conditions, one was to eliminate the overhead associated with the dynamic construction of the objects and virtual functions

and the other was to optimize the task graph along with these operations. To achieve this our framework constructs a simpler support structure that can link elements supported in click and import them to integrate with our application.

B. Parallel Processing Engine

After integrating the packet operations by means of support from `click`'s infrastructure and data access to network hardware through `netmap`'s APIs, we integrate these operations with the application graph, we now have a complete task graph that does both the computation and communication operations. The next logical step was executing these tasks in parallel. Two basic forms in which we can extract this lies in data and task parallelism. Packet streams have an inherent nature to operate in a data parallel fashion i.e. it is possible to operate on two different streams simultaneously. Since the application tasks in turn depend on the data packets they could also be executed in parallel. So from the integrated graph constructed by combining the application and packet processing operations it becomes possible to extract data parallelism by operating on different data streams simultaneously. In executing different graphs that represent distinct operations on the data streams it is possible to exploit task parallelism.

1) Execution engine: The execution engine consists of set of process and threads that the work is scheduled upon in order to be executed. When implementing threads to execute code which consists of system calls to the OS such as *malloc(), ioctl(), poll(),* etc., we found a degradation in performance even when the work that has to be executed is completely independent. To support the light weight style of threads the OS implements most of the locking mechanisms for system calls coming from threads. Previous research such as [17] [18] [19] 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)` [20] to construct a set of processes that is scheduled on to the different cpus on a multicore system.

In order to prevent interference from the OS scheduler's process migration, we specify *affinity* of the MPI process on to particular cpu cores. This prevents our process from being executed on a different cpu other than that specified. This makes our processes act as execution engines running on each of the cores on which we schedule the operations that are to be performed. Several previous research [21] [22] show the importance of affinity in network sensitive applications. Next, from utilizing our *Parallel Packet I/O* we specify the hardware queues each of these process handle. Initially we divide the queues evenly among the different processes we create, but later during execution we could modify the allocation based on the application operations scheduled on them. Our modifications to `netmap`'s hardware queue assignment API help us here in redistributing the queues to different processes. In using `MPI's send & receive` inter-process communication calls it makes it possible to modify the attributes of the process such as the cpus assigned or the number of queues it handles and even modify the operations the processes were initially assigned.

2) Parallel task processing: Once we have scheduled our execution engines on the physical system, the next step is to execute the integrated task graph consisting of application and packet operations on them. To extract data parallelism we replicate the task graph on to the multiple execution engines. This is to ensure operations on the same data are performed on the same cpu thereby reusing local cache. By making sure that packets belonging to specific flow always end up in the same execution engine, we only perform operations conforming to that stream of data. We leverage on the ability of modern network to classify packets, based on attributes of the packet, on to different queues.

```
bool payload_copy( const void *payload_data,
    uint32_t length )
{
    uint32_t headroom = _data - _head;
    uint32_t tailroom = _end - _tail;
    uint32_t orig_length = _tail - _data;
    uint32_t n = length + headroom + tailroom;
    if( n < min_buffer_length )
    {
        tailroom = min_buffer_length - length -
            headroom;
        n = min_buffer_length;
    }
    _data = _head + headroom;
    _tail = _data + length;
    if( payload_data )
        memcpy( _data, payload_data, length );
    return true;
}
```

Listing 1: Packet Payload Modification Function

In order to replicate the task graph we had to overcome some restrictions. First was to ensure that there was no intricate data dependency between different data streams. Although, click's library of elements are designed to be executed in parallel [12], it was more oriented towards being executed in a serial fashion. Hence we had to ensure the code present in the elements were thread safe. One important modification we had to perform was in the packet library that click uses. Click by default destroys and re creates the packet's internal storage buffer by reallocating memory space, which is an expensive operation. In order to avoid this overhead we allocate a packet with a large static buffer and copy the data to be modified on it. We modify the pointers that determine the header and data addresses based on the required packet length. This is done by using a simple sub routine shown in Listing 1 that we added to the click's packet library. Even though this incurs the overhead of a copy operation it does avoid performing invalid and dangerous memory operations directly on the network hardware queue's buffer.

Click's packet library does provide a mechanism to take ownership of a specific address and pass a destructor to be called when the packet is killed. To prevent corruption of memory mapped from the hardware queue by the application tasks we use the above mechanism to copy the data before

modifying it. In generating the application graph as a sequence of operations it also makes it possible to run different graphs on each individual execution engine. The MPI inter-process communication is used to schedule the application and packet operations on the processes representing the execution engines. Further more, we use it to also perform explicit data sharing on tasks in the application that requires it.

VI. Experimental Evaluation

The evaluation of our system should test the scalability and the performance speedup that we could achieve on a multicore system for an application that is constructed in our framework. To do this we choose two generic operations such as compression and encryption, built simple stream applications based on them in our framework. We then measured the improvement in performance as we increased the number of execution engines that are dedicated for the application.

A. Hardware Setup

The multicore system that is used to evaluate the framework consists of two servers connected directly to each other. Each of the server consists of two Intel Xeon 5600 hexacore HT in a dual socket configuration with 24GB DDR3 memory. So there is a total of 2 Processors x 6 Cores x 2 HT = 24 cpus in each server. These servers are connected with each other using Intel 8299EB 10Gbe adapters over direct attach twinaxial cables. They are placed in PCI Express x8 slot in each system to maximize the available bandwidth for the cards. This forms a closed connection between two servers and limits the influence of any external devices such as switches or routers on the actual data transmission.

The application is constructed as two parts, one acting as a source of data and other a sink. We measure the overall time it takes for application task operations, packet 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 considered to be the baseline in our experiments since this is similar to how the OS behaves for transmitting and receiving using the multi queue network controller.

The Intel 82599EB has limitations on the number of queues it can use for Receive Side Scaling (RSS) which is restricted to 16. So this limited our maximum amount of processes at the sink to be 16 even though there were 24 cpus available on the system. The source is not affected by this limitation and can scale to all the 24 cpus available on the system with the presence 24 hardware tx queues. Overall we at least have one hardware queue mapped to each of our processes or execution engines. We also ensure that each of the processes are bound to a particular cpu by issuing affinity call to the OS to ensure that we are not migrated between different cores.

B. Application Construction

In order to evaluate our framework we needed distinct operations that would be suitable to be applied in a streaming framework. Since compression and encryption applications are used commonly we chose two applications that represent these operations. The integrated graph that represents both these applications are shown in fig 3. Since our intention is to test the overall performance of the stream applications we use simple protocol such as udp in order to send data between the source and the sink.

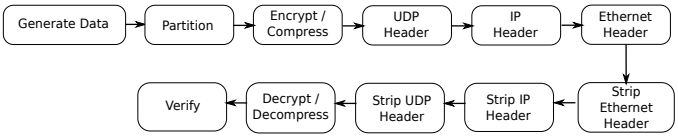

Fig. 3: Integrated Task Graph

Both the applications are constructed in a similar way. Two data sizes of 1.28 GB and 2.56 GB are selected as the workload for the compression and encryption applications respectively. Openssl library is used for the encryption and decryption and zlib is used for the compression and decompression operations respectively. The application consists of data generation phase after which it is compressed or encrypted depending on the application after which the packet operations are done. At this point the stream consisting of packets is sent over the network using the multiple queues. The sink side does the reverse of these operations. Once the data is received the packet operations are performed to extract the payload, after which the decompression or decryption operation is performed to retrieve the original data. These are some of the basic operations that would be required to construct a much more complex application. In choosing to compare these simple operations we avoid any limitations that might be present in a more complex application. This would enable us to evaluate the performance due to our framework and not that of any optimization done in the application.

VII. Results

We see that there is considerable difference in the way each of the operations behave as we scale them. This is shown in Figure 4. We can make three distinct observations from the results.

One is that from fig4.(d,e,f) we find that the compression requires a lot more work than that of decompression, whereas from fig4.(a,b,c) encryption and decryption seems to be more or less balanced. This is evident from the continued speedup we get up to 4 senders in the case of compression & decompression. In encryption & decryption this trend is not seen and the performance only improves by increasing both the number of senders and receivers. So we conclude that the combination of parallel tasks for application and packet operations differs between different applications. Determining this for each scenario is essential to boost the performance of the overall system.

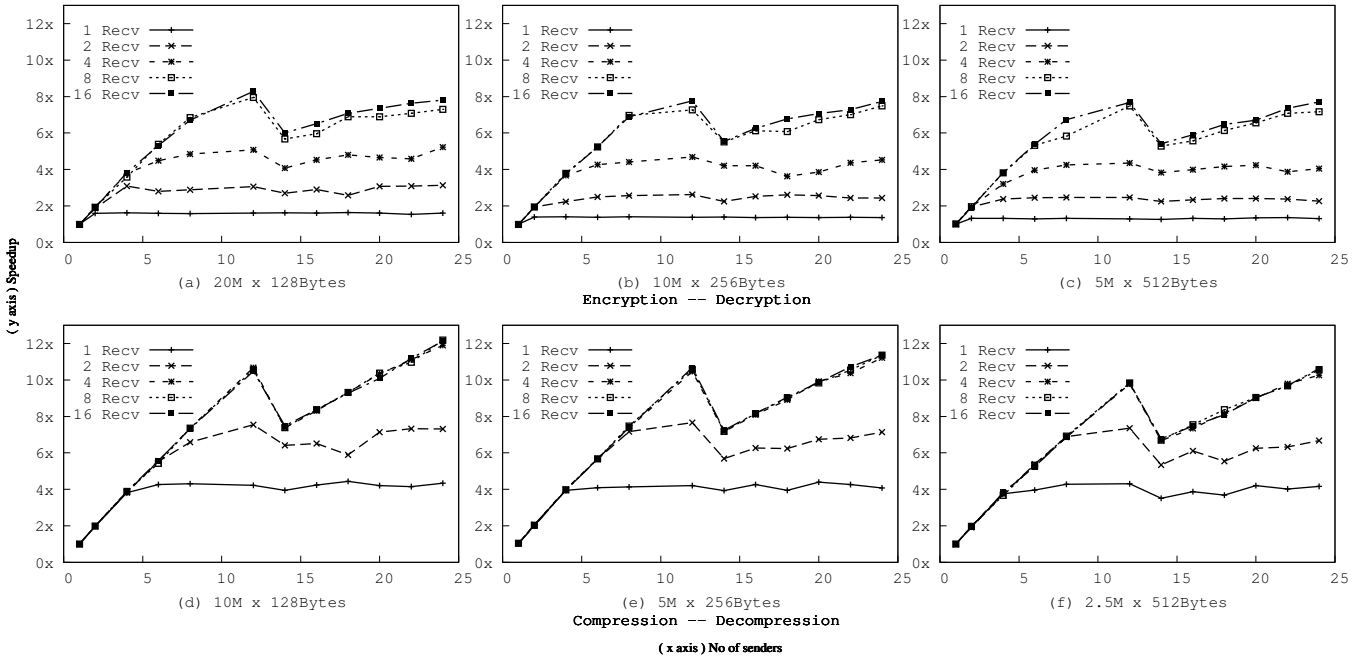

Fig. 4: Performance Speedup; (a),(b),(c) - Application A. Encryption; (d),(e),(f) - Application B. Compression; Speedup based on time in takes to compute a fixed workload for different number of source and sink processes and for varying packet sizes over a single source and sink are shown above

We see that the packet sizes have some impact on the performance from fig4.(d,e,f) but there is not much change compared to the overall speedup we achieve in each of the case. It can also be noticed that the compression operation seems to be influenced more than that of encryption operation. There seems to be speedup of 12x from fig4.(d) in the case of smaller packets and 10x from fig4.(f) for larger packets for compression operation. Whereas in the encryption application there is not much change and we get a speedup of 8x for any packet size observed from fig4.(a,b,c). While, packet sizes seem to have a much larger effect when there are smaller number of senders or receivers, but for larger cases not much change is observed in fig4.(d,e,f). So we conclude that the influence of a parameter for the network system such as a packet size varies between different applications and it is essential to have a unified system that can tackle both the operations together.

From fig4(a,b,c,d,e,f), we noticed the drop in speedup as we move from 12 senders to more. This could be due to the influence of NUMA architecture [23] on the performance of the system. This is the particular point in which we utilize all the cores of a particular cpu and move on to a different socket. Mainly, the combination of cache locality and limitation of the network controller to receive data through the PCI-E bus from both the processors could be the contributing factors. Still, we find that even in a NUMA architecture once our framework overcomes the initial drop, it continues to improve and eventually attain better performance in most cases.

We can see that our framework scales very well for the different workloads provided and can handle both the computations and communication and operate on them in parallel. In some cases we find that our framework could be improved by having a much more control over the fine grained parallelism within the execution engines. This could be done by introducing pipeline stages for the different operations.

VIII. Related Work

Several prior research [6] [7] [24] [25] have worked on using commodity multicore processors to perform network operations.

Dobrescu et al. [6] shows the performance improvement achieved in the construction of software routers by distributing the related operations over multiple servers.

Han et al. [7] perform tasks related to high performance packet processing on commodity hardware using GPUs. They have developed a novel framework know as PacketShader that can do packet processing in user space by exploiting massively-parallel architecture of GPUs.

Wang et al. [24] constructs a high performance connection-affinity based lock-free multicore network processing system that claims to achieve multiple Gbps network processing speed for complex tasks.

Egi et al. [25] shows techniques on parallelizing network operations on multicore architectures by defining principles that consider multiple resources and not just the CPU alone.

Stream programming languages such as StreamIt [1], LUSTRE [2], etc., target stream applications but they focus more on parallelizing the computations related to the application alone on the multicore system.

`Netmap` [11], provides the APIs to access the hardware queue's buffer in user space by means of memory mapped

rings. It implements an intermediate ring in kernel space that is used to synchronize the data from the user space to that of hardware queues in the NIC.

Click modular router [8] is an extensively used tool for the construction of software routers using general purpose hardware and provides a massive infrastructure to support this. SMP Click [12] is an improved version, which supports parallel processing of these using threads. Click implements its own mechanism and polling driver in kernel space to access packets, bypassing the operating system's network infrastructure.

IX. Conclusions

In this paper we have proposed a high-level programming framework that can integrate packet processing within stream applications and perform both the operations in parallel.

We conclude that by using our algorithms for sending and receiving data, which implicitly performs batching and reduces the usage of system calls, and in combining this with `netmap` APIs which minimizes the interaction with the operating system, communication performance in our framework never becomes the bottleneck in our system on the programs we have tested. Instead of performing the packet operations in the OS, by integrating it with the application using click's elements within our infrastructure we are able to construct a unified task graph that represents the computation and communication operations. Using this we are able to parallelize these operations together and achieve speedups of more than a factor of eight in all the applications we tested.

In integrating these features into a programming 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. The results shows that our system scales well to as many as 24 parallel processes on a multicore computer, and achieves speedups of more than a factor of ten in some cases compared to sequential versions of the stream applications.

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. [Online]. Available: http://dl.acm.org/citation.cfm?id=647478.727935

[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," 1995.

[4] W. Wu and M. Crawford, "Potential performance bottleneck in linux tcp," *Int. J. Commun. Syst.*, vol. 20, no. 11, pp. 1263–1283, Nov. 2007. [Online]. Available: http://dx.doi.org/10.1002/dac.v20:11

[5] 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. [Online]. Available: http://dx.doi.org/10.1016/j.comcom.2006.11.001

[6] 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.

[7] 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. [Online]. Available: http://doi.acm.org/10.1145/1851275.1851207

[8] 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. [Online]. Available: http://doi.acm.org/10.1145/354871.354874

[9] (2008, Nov.) Microsoft corporation, receive-side scaling enhancements in windows server 2008,. [Online]. Available: http://download.microsoft.com/ download/ a/d/f/adf1347d-08dc-41a4-9084-623b1194d4b2 RSS Server2008.docx

[10] (2010, Nov.) Intel corporation, intel 82599 10gbe controller datasheet. [Online]. Available: http://download.intel.com/design/network/datashts82599_datasheet.pdf

[11] L. Rizzo, "Revisiting network i/o apis: The netmap framework," *Queue*, vol. 10, no. 1, pp. 30:30–30:39, Jan. 2012. [Online]. Available: http://doi.acm.org/10.1145/2090147.2103536

[12] B. Chen and R. Morris, "Flexible control of parallelism in a multiprocessor pc router," in *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2001, pp. 333–346. [Online]. Available: http://dl.acm.org/citation.cfm?id=647055.759948

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

[14] M. Krasnyansky, "Uio-ixgbe." [Online]. Available: https://opensource.qualcomm.com/wiki/UIO- IXGBE

[15] Libnids. [Online]. Available: http://libnids.sourceforge.net

[16] A. Dunkels, "Design and implementation of the lwip," 2001.

[17] 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. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267724.1267780

[18] D. Dice and A. Garthwaite, "Mostly lock-free malloc," *SIGPLAN Not.*, vol. 38, no. 2 supplement, pp. 163–174, Jun. 2002. [Online]. Available: http://doi.acm.org/10.1145/773039.512451

[19] M. M. Michael, "Scalable lock-free dynamic memory allocation," *SIGPLAN Not.*, vol. 39, no. 6, pp. 35–46, Jun. 2004. [Online]. Available: http://doi.acm.org/10.1145/996893.996848

[20] C. 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. [Online]. Available: http://doi.acm.org/10.1145/169627.169855

[21] 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. [Online]. Available: http://dx.doi.org/10.1109/90.532862

[22] 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.

[23] 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. [Online]. Available: http://doi.acm.org/10.1145/1854273.1854314

[24] J. Wang, H. Cheng, B. Hua, and X. Tang, "Practice of parallelizing network applications on multi-core architectures," in *Proceedings of the 23rd international conference on Supercomputing*, ser. ICS '09. New York, NY, USA: ACM, 2009, pp. 204–213. [Online]. Available: http://doi.acm.org/10.1145/1542275.1542307

[25] N. Egi, G. Iannaccone, M. Manesh, L. Mathy, and S. Ratnasamy, "Improved parallelism and scheduling in multi-core software routers," *The Journal of Supercomputing*, pp. 1–29, 10.1007/s11227-011-0579-3. [Online]. Available: http://dx.doi.org/10.1007/s11227-011-0579-3