Inferring the Topology and Traffic Load of Parallel Programs Running In a Virtual Machine Environment

Ashish Gupta and Peter A. Dinda
{ashish,pdinda}@cs.northwestern.edu

Department of Computer Science, Northwestern University

Abstract. We are developing a distributed computing environment based on virtual machines featuring application monitoring, network monitoring, and an adaptive virtual network. In this paper, we describe our initial results in monitoring the communication traffic of parallel applications, and inferring its spatial communication properties. The ultimate goal is to be able to exploit such knowledge to maximize the parallel efficiency of the running parallel application by using VM migration, virtual overlay network configuration and network reservation techniques, which are a part of the distributed computing environment. Specifically, we demonstrate that: (1) we can monitor the parallel application network traffic in our layer 2 virtual network system with very low overhead, (2) we can aggregate the monitoring information captured on each host machine to form a global picture of the parallel application's traffic load matrix, and (3) we can infer from the traffic load matrix the application topology. In earlier work, we have demonstrated that we can capture the time dynamics of the applications. We begin here by considering offline traffic monitoring and inference as a proof of concept, testing it with a variety of synthetic and actual workloads. Next, we describe the design and implementation of our online system, the Virtual Topology and Traffic Inference Framework (VTTIF), and evaluate it using a NAS benchmark.

1 Introduction

Virtual machines have the potential to simplify the use of distributed resources in a way unlike any other technology available today, making it possible to run diverse applications with high performance after only minimal or no programmer and administrator effort. Network and host bottlenecks, difficult placement decisions, and firewall obstacles are routinely encountered, making effective use of distributed resources an obstacle to innovative science. Such problems, and the human effort needed to work around them, limit the development, deployment, and scalability of distributed parallel applications.

We have presented a detailed case for virtual machine-based distributed and parallel computing [4], and we are now developing a system, Virtuoso, which has the following model:

Effort sponsored by the National Science Foundation under Grants ANI-0093221, ACI-0112891, ANI-0301108, EIA-0130869, and EIA-0224449. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation (NSF).

– The user receives what appears to be a new computer or computers on his network at very low cost. The user can install, use, and customize the operating system, environment, and applications with full administrative control.
– The user chooses where to execute the virtual machines. Checkpointing and migration is handled efficiently through Virtuoso. The user can delegate these decisions to the system.
– A service provider need only install the VM management software to support a diverse set of users and applications.
– Monitoring, adaptation, resource reservation, and other services are retrofitted to existing applications at the VM level with no modification of the application code, resulting in broad application of these technologies with minimal application programmer involvement.

An important element of our system is a layer 2 virtual network, VNET, which we initially developed to create the "networking illusion" needed for the first element of the model. It can "move" a set of virtual machines in a WAN environment to the user's local layer 2 domain. We are now expanding VNET into a tool that supports arbitrary overlay topologies and routing rules, passive application and network monitoring, adaptation (based on VM migration and topology/routing changes), and resource reservation. VNET is described in detail in a previous paper [12].

This paper reports on one of our first steps toward achieving the last element of the model. The question we address in particular is: can we monitor, with low overhead and no application or operating system modifications, the communication traffic of a parallel application running in a set of virtual machines interconnected with a virtual network, and compute from it the traffic load matrix and application communication topology? Our initial results demonstrate that this is possible. We are integrating the online implementation of our ideas, VTTIF (Virtual Topology and Traffic Inference Framework), into the evolving VNET system.

We consider here Bulk-Synchronous Parallel [6] (BSP) style applications. Specifically, we consider parallel programs whose execution alternates between one or more computing phases and one or more communication phases, including metaphases. We are testing whether our results hold for more general applications. In earlier work, we have demonstrated that the network traffic of compiler-parallelized BSP applications, when measured using techniques similar to those used here, exhibits clear time dynamical structure (periodicity with harmonics) [3]. Our results here show that we can quickly and efficiently recover its spatial structure, its topology and traffic load, as well.

The ultimate motivation behind recovering the spatial and temporal properties of a parallel application running in a virtual environment is to be able to maximize the parallel efficiency of the running application by migrating its VMs, changing the topology and routing rules of the communication network, and taking advantage of underlying network reservations on the application's behalf.

A parallel program may employ various communication patterns for its execution. A communication pattern consists of a list of all the message exchanges of a representative processor during a communication phase. The result of each processor executing

its communication pattern gives us the application topology, such as a mesh, toroid, hypercube, tree, etc, which is in turn mapped to the underlying network topology [7]. In this paper, we attempt to infer the application topology and the costs of its edges, the traffic load matrix, by observing the low-level traffic entering and leaving each node of the parallel application, which is running inside of a virtual machine.

It is important to note that application topologies may be arbitrarily complex. Although our initial results are for BSP-style applications, our techniques can be used with arbitrary applications, indeed, any application or OS that the virtual machine monitor (we use VMWare GSX server in this work) can support. However, we do not yet know the effectiveness of our load matrix and topology inference algorithms for arbitrary applications.

In general, it is difficult for an application developer, or, for that matter, the user of a "dusty deck" application, to analyze and describe his application at the level of detail needed in order for a virtual machine distributed computing system to make adaptation decisions on its behalf. Furthermore, the description may well be time or data dependent or react to the conditions of the underlying network.

The goal of VTTIF is to provide these descriptions automatically, as the unmodified application runs on an unmodified operating system. In conjunction with information from other monitoring tools, and on the policy constraints, VTTIF information will then be used to schedule the VMs, migrate them to appropriate hosts, and change the virtual network connecting them. The adaptation control mechanisms will query VTTIF to understand what, from a communication perspective, the parallel application is attempting to accomplish.

We began by offline analysis, using traffic logs of parallel applications to develop our three step monitoring and analysis process. Although this initial work was carried out without the use of VMs, using PVM applications whose traffic was captured using tcpdump techniques, it is directly applicable for two reasons. First, VNET interacts with the virtual interfaces of virtual machines in a manner identical (packet filter on the virtual interface) to how tcpdump interacts with physical interfaces (packet filter on a physical interface). Second, the physical machines generate considerably more "noise" than the virtual machines, thus making the problem harder. In Section 2, we describe our three step process and how it is implemented for physical monitoring. In Section 3 we describe a set of synthetic applications and benchmarks we will use to evaluate VTTIF. In Section 4, we show the performance results of applying the process to a wide variety of application topologies and parallel benchmarks.

The results for the offline, physical machine-based were extremely positive, so we designed and implemented an online process that is integrated with our VNET virtual networking tool. Section 5 describes the design of the online VTTIF tool and provides an initial evaluation of it. We are able to recover application topologies online for a NAS benchmark running in VMs and communicating via VNET. The performance overhead of the VTTIF implementation in VNET is negligible.

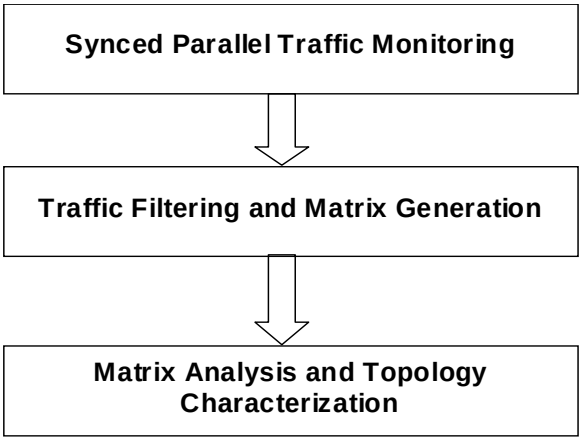

Fig. 1. The three stages involved in inferring the topology and traffic load matrix of a parallel application

In Section 6, we conclude by describing our plans for using the VTTIF and other monitoring information for heuristic adaptive control of the VMs and VNET to maximize application performance.

2 VTTIF and its offline implementation

The inference of parallel application communication is based on the analysis of low level traffic. We first wanted to test whether this approach was practical at all, and, if so, to develop an initial framework for traffic monitoring, analysis and inference, enabling us to test our ideas and algorithms. This initial step resulted in an offline process that focused on parallel programs running on physical hosts. In Section 5, we describe how these results have been extended to an online process that focuses on parallel programs running in virtual machines.

In both our online and offline work, we study PVM [5] applications. Note that the techniques described here are general and are also applicable to other parallel applications such as MPI programs. We run programs on the nodes of our Virtuoso cluster, which is an IBM e1350 with 32 compute nodes, each of which is a dual 2.2 GHz Intel HT Xeon Processors, 1.5 GB RAM, and 40 GB of disk. Each node runs Red Hat Linux 9, PVM 3.4.4, and VMWare GSX Server 2.5. Each VM runs Red Hat Linux 7.3 and PVM 3.4.4. The communication measured here is via a 100 mbit switched network, specifically a Cisco 3550 48 port switch. The nodes speak NFS and NIS back to a separate management machine via a separate network.

The VTTIF framework has three stages as shown in Figure 1. In the first stage, we monitor the traffic being sourced and sinked by each process in the parallel program.

In the offline analysis, this is accomplished by using tcpdump on the physical interface with a packet filter that rejects all but PVM traffic. In the online analysis, we integrate monitoring into our virtual network tool VNET. VNET does the equivalent of running tcpdump on the virtual interface of the virtual machine, capturing all traffic. The Virtuoso cluster uses a switched LAN, so the interface of each node must be monitored separately and the data aggregated. A challenge in the online system is that it must decide when to start and stop this monitoring.

The second stage of the framework eliminates irrelevant traffic from the aggregated traffic and integrates the packet header traces captured by tcpdump to produce a traffic matrix, T. Element $T_{i,j}$ represents the amount of traffic sent from node i to node j. A challenge in the online system is to decide when to recompute this matrix.

The final stage of the framework applies inference algorithms to eliminate noise from the traffic matrix to infer from it the likely application topology. Both the original matrix and the inferred topology are then returned. The topology is displayed graphically. A challenge in the online system is to decide when to recompute the topology.

The current offline framework is designed to automate all of the above steps, allowing the user to run a single command, `infer [parallel PVM program]` This runs the PVM program mentioned in the argument, monitors it for its entire execution, completes the remaining steps of the framework, and prints the matrix and displays the topology. The framework is implemented as a set of Perl scripts, as described below.

Monitor This script is responsible for synchronized traffic monitoring on all the physical hosts, running the parallel program, and storing the packet header traces to files. The script also reads a configuration file that describes the set of hosts on which monitoring is to be done. It runs tcpdump on each of the hosts. It then executes the parallel program and waits for it to finish. Each tcpdump stores its packet header trace output into a file named by the hostname, the date, and the time of execution. Hence, each execution produces a group of related packet header trace files.

Generate This script parses, filters and analyzes the packet header traces to generate a traffic matrix for the given hosts. It sums the packets sizes between each pair of hosts, filtering out irrelevant packets. Filtering is done according to the following criteria:

- Type of packet. Packets which are known not to be a part of the parallel program communication, like ARP, X11, ssh, etc, are discarded. This filtering has only a modest effect in the Virtuoso cluster because there is little extra traffic. However, in the future, we may want to run parallel programs in a wide area environment or one shared with many other network applications, where filtering and extracting the relevant traffic may pose extra challenges.
- The source and destination hosts involved in the packet transmission. We are only interested traffic among a specific group of hosts.

The matrix is emitted in a single file.

Infer This script infers the application topology from the traffic matrix file. In effect, topology inference amounts to taking the potentially "noisy" graph described by the traffic matrix and eliminating edges that are unlikely to be significant. The script also outputs a version of the topology that is designed to be viewed by the algorithm animation system Samba [11].

For inferring the topology, various algorithms are possible. One method is to prune all matrix entries below a certain threshold. More complex algorithms could employ pattern detection techniques to choose an archetype topology that the traffic matrix is most similar to. For the results we show, topology inference is done using a matrix normalization and simple edge pruning technique. The pseudo-code description of the algorithm is:

InferTopology(traffic_matrix T,pruning_threshold b_{min}*)*
{
    $b_{max} \leftarrow max(T_{i,j}) \; \forall_{i,j}$
    $G \leftarrow \emptyset$
    foreach($T_{i,j}$)
    {
        $r_{i,j} \leftarrow T_{i,j}/b_{max}$
        if ($r_{i,j} \geq b_{min}$)
        {
            add edge(i,j) to G
        }
    }
    return G
}

In effect, if the maximum bandwidth entry in T is b_{max}, then if ratio of any edge value ($T_{i,j}$) to b_{max} is below a certain threshold b_{min}, then the edge is pruned. The value of b_{min} determines the sensitivity of topology inference.

Visualization makes it very convenient to quickly understand the topology used by the parallel program. By default, we have Samba draw each topology with the nodes laid out in a circle, as this is versatile for a variety of different topologies. However, there is an option to pass a custom graph layout. An automated layout tool such as Dot could also be used.

Figure 2 shows an example of the final output for the program PVM POV, a parallel ray tracer, running on four hosts. The thickness of an edge indicates the amount of traffic for that particular run. Each host is represented by a different color and color of the edge represents the source host for the edge traffic.

3 Workloads for VTTIF

To test our ideas, we first needed some actual parallel applications to measure. We created and collected the following applications.

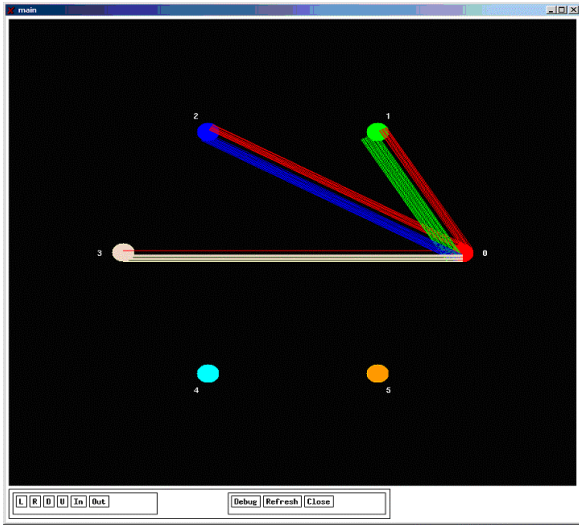

Fig. 2. An example of the final output of the Topology Inference Framework for the PVM-POV application. The PVM-POV application runs on four hosts.

- Patterns: This is a synthetic workload generator, which we describe below. It can execute many different kinds of topologies common in BSP parallel programs. We use this extensively to test our framework.
- NAS Parallel Benchmarks: We use the PVM implementation of the NAS benchmarks [1] IS, MG, FT, and EP as developed by Sundaram, et al [13].
- PVM POV: PVM version of the popular ray tracer POVRAY. The PVM version gives it the ability to distribute a rendering across multiple heterogeneous systems. [2].

Except for patterns, these are all well known benchmark programs.

Patterns does message exchanges according to a topology provided at the command line. Patterns emulates a BSP program with alternating dummy compute phases and communication phases according to the chosen topology. It takes the following arguments:

- pattern: The particular topology to be used for communication.
- numprocs: The number of processors to use. The processors are determined by a special hostfile.
- messagesize: The size of the message to exchange.
- numiters: The number of compute and communicate phases
- flopsperelement: The number of multiply-add steps
- readsperelement: The number of main memory reads
- writesperelement: The number of main memory writes

Patterns generates a deadlock free and efficient communication schedule at startup time for the given topology and number of processors to be used. The following topologies are supported:

- n-dimensional mesh, neighbor communication pattern
- n-dimensional torus, neighbor communication pattern
- n-dimensional hypercube, neighbor communication pattern
- Binary reduction tree
- All-to-all communication

4 Evaluation of offline VTTIF

We evaluated our offline inference framework with the various parallel benchmarks described in the previous section. Figure 3 shows the inferred application topologies of various patterns benchmark runs, as detected by our offline framework. These results suggest that there is indeed considerable promise in traffic-based topology inference: parallel program communication behavior can be inferred without any knowledge of the parallel application itself. Of course, more complex filtering processes may need to be used for more complex applications and complex network environments where parallel application traffic is just a part of the network traffic.

We also ran the application benchmarks described earlier. These results are also promising. Figure 4 shows a representative, the traffic matrix for an execution of the Integer Sort (IS) NAS kernel benchmark on 8 physical hosts, with the corresponding topology shown in Figure 5. The topology resembles an all-to-all communication, but the thickness of the edges vary indicating that the bandwidth requirements vary depending on the host pairs. A closer look at the traffic matrix reveals that $host_1$ receives data in the range of 20 MB from each of the other hosts, indicating that this is a communication intensive benchmark. Other hosts $host_2$ to $host_8$ transfer data of $\simeq 10 - 11$ MB with each other, almost half of that exchanged with $host_1$.

Notice that this information could be used to boost the performance of the IS benchmark if it were running in our VM computing model. Ideally, we would move the VM $host_1$ to a host with relatively high bandwidth links and reconfigure the virtual network with appropriate virtual routes over the physical network [10]. Such decisions need to be dynamic, as the properties of a physical network vary [14]. Without any intervention by the application developer or knowledge of the parallel application itself, it is feasible to infer the spatial and temporal [3] properties of the parallel application. Equipped with this knowledge, we can use VM checkpointing and migration along with VNET's virtual networking capabilities to create a efficient network and host environment for the application.

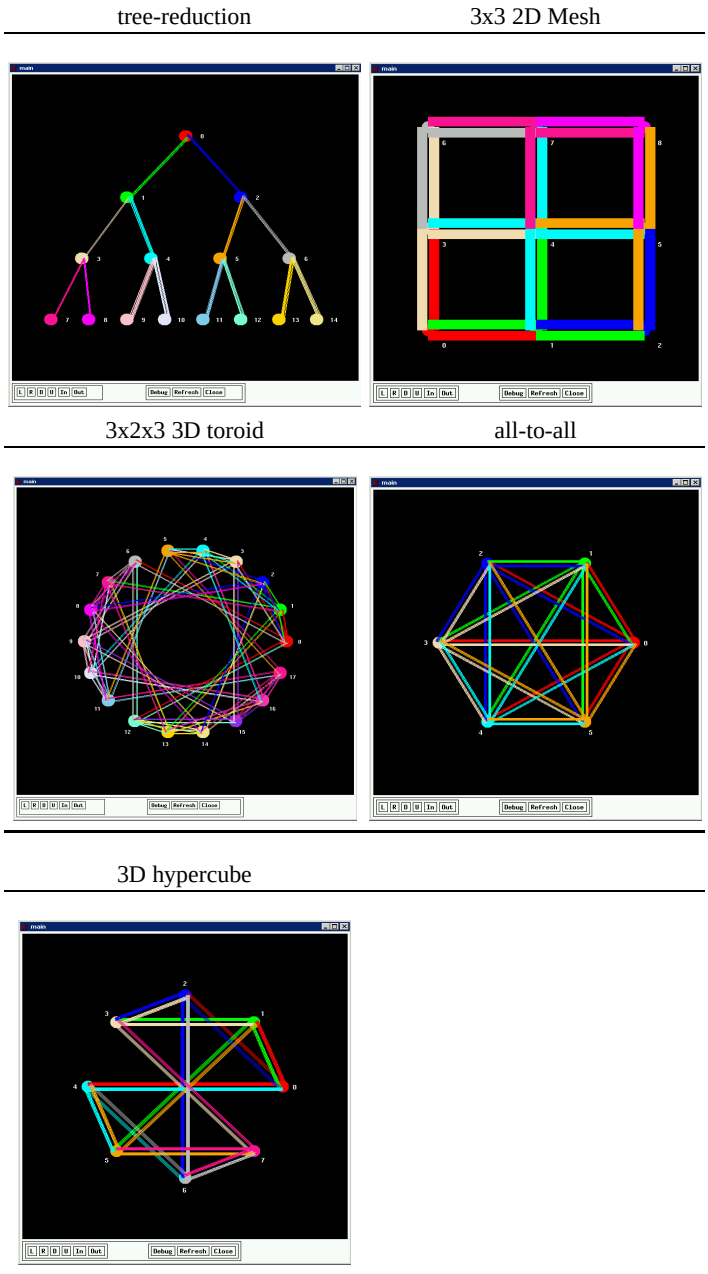

Fig. 3. The communication topologies inferred by the framework from the patterns benchmark. It shows the inferred tree-reduction, 3x3 2D Mesh, 3x2x3 3D toroid, all-to-all for 6 hosts and 3D hypercube topologies.

	h1	h2	h3	h4	h5	h6	h7	h8
h1		19.0	19.6	19.2	19.6	18.8	13.7	19.3
h2	22.6		10.7	10.8	10.7	10.9	9.7	10.5
h3	22.2	8.78		11.2	10.4	10.1	10.5	10.5
h4	22.4	8.9	9.5		11.1	10.8	10.6	10.2
h5	22.3	10.0	9.51	9.72		11.7	10.9	11.9
h6	24.0	8.9	10.7	9.9	10.8		12.2	12.1
h7	23.2	10.0	9.7	9.5	10.3	10.2		12.0
h8	24.9	11.2	11.0	11.8	11.5	11.2	10.7	

*numbers indicate MB of data transferred.

Fig. 4. The traffic matrix for the NAS IS kernel benchmark on 8 hosts.

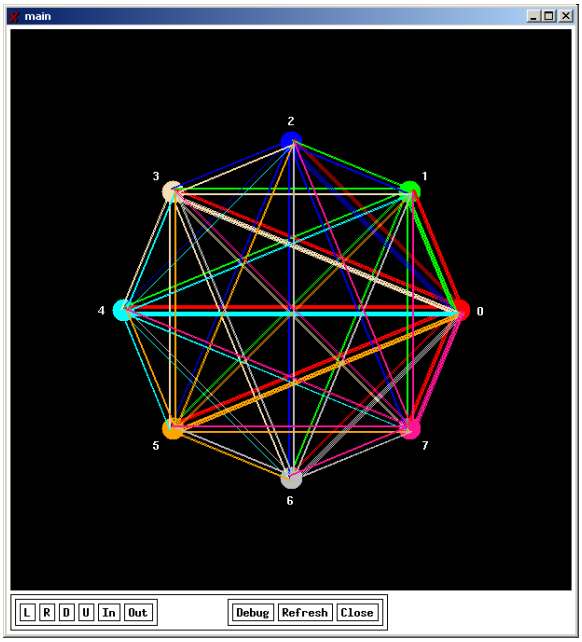

Fig. 5. The inferred topology for the NAS IS kernel benchmark

5 Online VTTIF

After working with offline parallel program topology inference on the physical hosts, the next step was to develop an online framework for a virtual machine environment. We extended VNET [12], our virtual networking tool, to include support for traffic analysis and topology inference. VNET allows the creation of layer 2 virtual networks interconnecting VMs distributed over an underlying TCP/IP networking infrastructure. A VNET daemon manages all the network traffic of the VMs running on its host, and thus is an excellent place to observe the application's network traffic. All traffic monitoring is done at layer 2, providing flexibility in analyzing and filtering the traffic at many layers.

Due to a networking issue with the Virtuoso cluster, the work in the section was done on a slower cluster consisting of dual 1 GHz Pentium III processors with 1 GB of RAM and 30 GB hard disks. We used a switched 100 mbit network connecting the machines. As before VMWare GSX Server 2.5 was used, except here it was run on Red Hat Linux 7.3. The VMs were identical. A Dell PowerEdge 4400 (dual 1 GHz Xeon, 2 GB, 240 GB RAID) running Red Hat 7.1 was used as the VNET proxy machine.

5.1 Observing traffic phenomena of interest: reactive and proactive mechanisms

VMs can run for long periods of time, but their traffic may change dramatically over time as they run multiple applications in parallel or serially. In the offline VTTIF, monitoring and aggregation are triggered manually while running the parallel application. This is not possible in an online design. The online VTTIF needs a mechanism to detect and capture traffic patterns of interest, reacting automatically to interesting changes in the communication behavior of the VMs. It must switch between active states, when it is accumulating data and computing topologies, and passive states, when it is waiting for traffic to intensify or otherwise become relevant. Ideally, VTTIF would have appropriate information available whenever a scheduling agent requests it.

We have implemented two mechanisms for detecting interesting dynamic changes in communication behavior: reactive and proactive. In the reactive mechanism, VTTIF itself alerts the scheduling agent when it detects certain pre-specified changes in communication. For example, in the current implementation, VTTIF monitors the rate of traffic for all flows passing through it and starts aggregating traffic information whenever the rate crosses a threshold. If this rate is sustained, then VTTIF can alert the scheduling agent about this interesting behavior along with conveying its local traffic matrix.

In the proactive mechanism, VTTIF allows an external agent to make traffic-related queries such as: *what is traffic matrix for the last 512 seconds?* VTTIF stores sufficient history to answer various queries of interest, but it does not alert the scheduling agent, unlike the reactive mechanism. The agent querying traffic information can determine its own policy, for example polling periodically to detect any traffic phenomena of interest and thus making appropriate scheduling, migration and network routing decisions

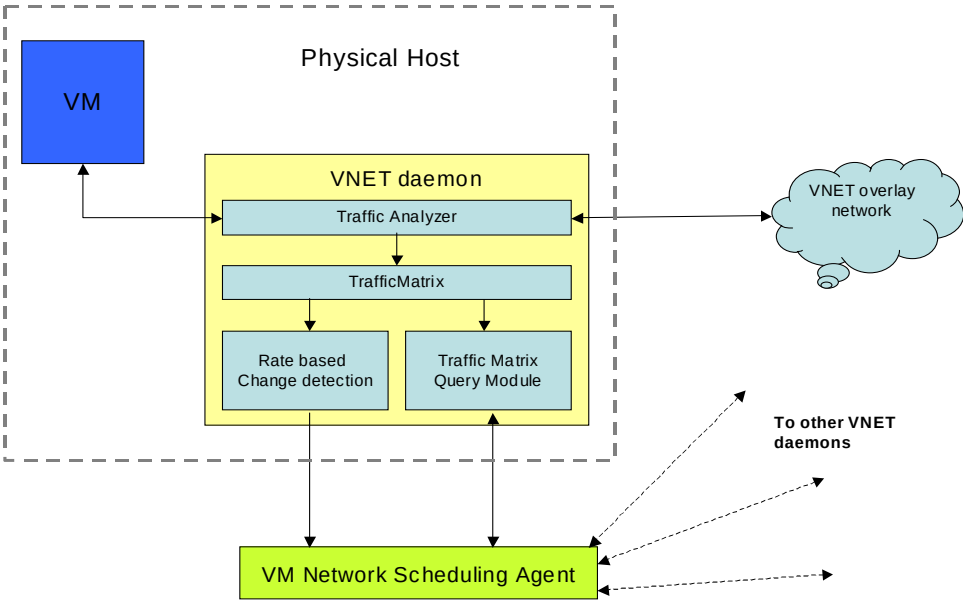

Fig. 6. The VNET-VTTIF topology inference architecture. VTTIF provides both reactive and proactive services for the scheduling agent.

to boost parallel application performance. Figure 6 shows the high level view of the VNET-VTTIF architecture. The VM and overlay network scheduling agent may be located outside the VM-side VNET daemon, and all relevant information can conveyed to it so that it can make appropriate scheduling decisions.

5.2 Implementation

We extended VNET so that each incoming and outgoing Ethernet packet passes through a packet analyzer module. This function parses the packet into protocol headers (Ethernet, IP, TCP) and can filter it if it is irrelevant. Currently all non-IP packets are filtered out—additional filtering mechanisms can be installed here. Packets that are accepted are aggregated into a local traffic matrix. Specifically, for each flow, a row and column of the matrix are determined in this way. The matrix is stored in a specialized module TrafficMatrix. TrafficMatrix is invoked on every packet arrival.

Reactive mechanism The TrafficMatrix module does non-uniform discrete event sampling for each source/destination VM pair to infer the traffic rate between the pair. The functioning of `rate_threshold` mechanism is illustrated in Figure 7. It takes two parameters: `byte_threshold` and `time_bound`. Traffic is said to cross the

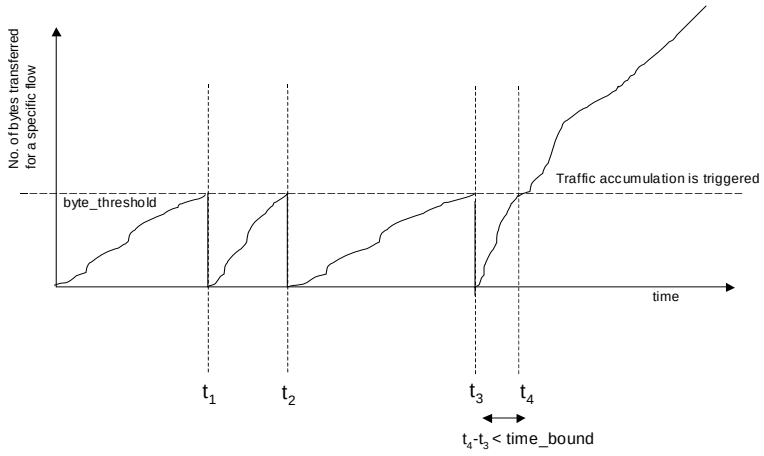

Fig. 7. The rate-threshold detection based reactive mechanism in VNET-VTTIF. Whenever two successive byte thresholds are exceeded within a time bound, the accumulation of traffic is triggered.

`rate_threshold`, if for a particular VM pair, `byte_threshold` bytes of traffic is transmitted in a time less than `time_bound`. This is detected by time-stamping the packet arrival event whenever the number of transmitted bytes for a pair exceeds a integral multiple of `byte_threshold`. If two successive time-stamps are less than `time_bound`, this indicates our `rate_threshold` requirement has been met. Once a pair crosses the `rate_threshold`, TrafficMatrix starts accumulating traffic information for all the pairs. Before the `rate_threshold` is crossed, TrafficMatrix doesn't accumulate any information, i.e. it is a memoryless system. After the `rate_threshold` is crossed, TrafficMatrix alerts the scheduling agent in two situations. First, if the high traffic rate is sustained up to time t_{max}, then it sends all its traffic matrix information to the scheduling agent. In other words, TrafficMatrix informs the scheduling agent if an interesting communication behavior persists for a long enough period of time. The second situation is if the rate falls below the threshold and remains there for more than t_{wait} seconds, in which case TrafficMatrix alerts the scheduling agent that the application has gone quiet.

Figure 8 illustrates the operation of the reactive mechanism in flowchart form.

Proactive mechanism The proactive mechanism allows an external agent to pose queries to VTTIF and then take decisions based on its own policy. VTTIF is responsible solely for providing the answers to useful queries. TrafficMatrix maintains a history for all pairs it is aware in order to answer queries of the following form: What is the traffic matrix over the last n seconds? To do so, it maintains a circular buffer for all pairs in which each entry corresponds to the number of bytes transferred in a particular second. As every packet transmission is reported to TrafficMatrix, it updates the circular buffer

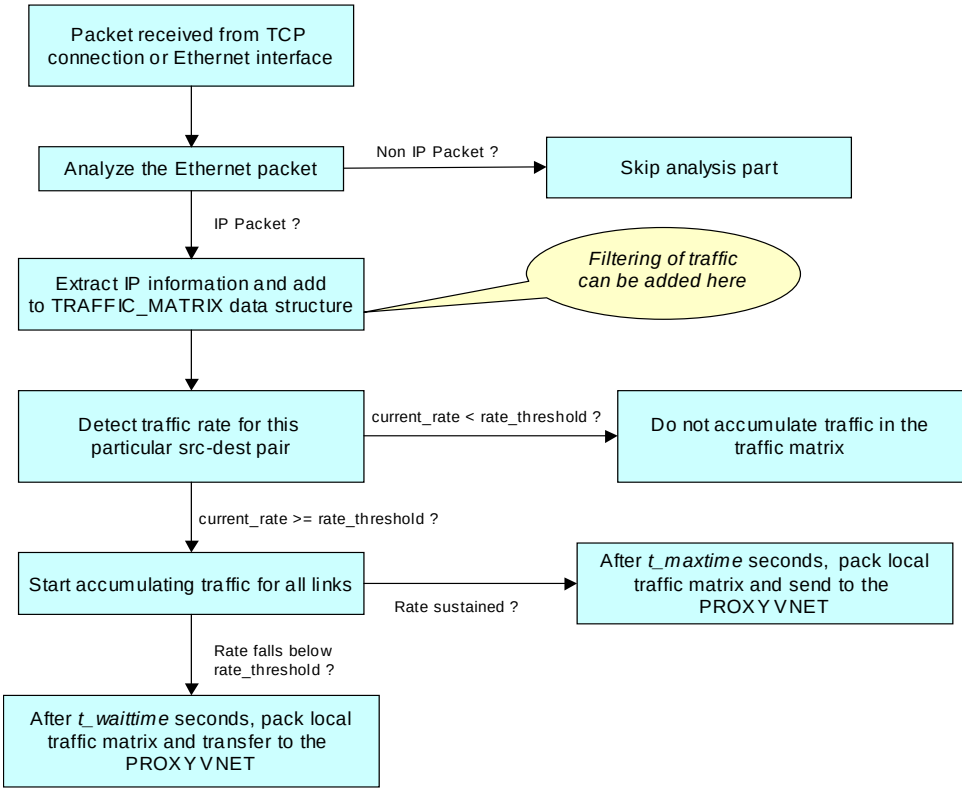

Fig. 8. The steps taken in the VM-side VNET daemon for the reactive mechanism.

for the particular pair. To answer the query, the last n entries are summed up, up to the size of the buffer.

The space requirements for storing the state history needs some consideration. The space requirements depends on the maximum value of n. For each pair, $4n$ bytes are needed for the circular buffer. If there are m VMs, then the total space allocation is $4nm^2$. For $n = 3600$ (1 hour) and $m = 16$ VMs, the worst case total space requirement is 3.7 Mbytes. A sparse matrix representation could considerably reduce this cost and thus the communication cost in answering the queries.

5.3 Aggregation

Aggregation of traffic matrices from the various VNET daemons provides a global view of the communication behavior exhibited by the VMs. Currently, we aggregate the locally collected traffic matrices to a global, centralized matrix that is stored on the VNET proxy daemon, which is responsible for managing the virtual overlay network in VNET.

Method	Average	STDEV	Min	Max
Direct	0.529 ms	0.026 ms	0.483 ms	0.666 ms
VNET	1.563 ms	0.222 ms	1.277 ms	2.177 ms
VNET-VTTIF	1.492 ms	0.198 ms	1.269 ms	2.218 ms

Fig. 9. Latency comparison between VTTIF and other cases

We use a push mechanism—the VNET daemons decide when to send their traffic matrix based on their reactive mechanism. A pull mechanism could also be provided, in which the proxy would request traffic matrices when they are needed based on queries.

The storage analysis of the previous sections assumes that we will collect a complete copy of the global traffic matrix on each VNET daemon—in other words, that we will follow the reduction to the proxy VNET daemon with a broadcast to the other VNET daemons. This is desirable so that the daemons can make independent decisions. However, if we desire only a single global copy of the whole matrix, or a distributed matrix, the storage and computation costs will scale with the number of VMs hosted on the VNET daemon.

Scalability is an issue in larger instances of the VM-based distributed environment. Many possibilities exist for decreasing the computation, communication and storage costs of VTTIF. One optimization would be to maintain a distributed traffic matrix. Another would be to implement reduction and broadcast using a hierarchical structure, tuned to the performance of the underlying network as in ECO [8]. Fault tolerance is also a concern that needs to be addressed.

5.4 Performance overhead

Based on our measurements, VTTIF has minimal impact on bandwidth and latency. We considered communication between two VMs in our cluster, measuring round-trip latency with ping and bandwidth with ttcp. Figure 9 compares the latency between the VMs for three cases:

- Direct communication. Here VNET is not involved. The machines communicate locally using VMWare's bridged networking. This measures the maximum performance achievable between the hosts, without any network virtualization.
- VNET. Here we use VNET to proxy the VMs to a different network through the PowerEdge 4400. This shows the overhead of network virtualization. Note that we are using an initial version of VNET here without any performance enhancements running on a stock kernel. We continue to work to make VNET itself faster.
- VNET-VTTIF. This case is identical to VNET except that we are monitoring the traffic using VTTIF.

There is no significant difference between the latency of VNET and VNET-VTTIF.

Method	Throughput
Direct	11485.75 KB/sec
VNET	8231.82 KB/sec
VNET-VTTIF	7895.06 KB/sec

Fig. 10. Throughput comparison between VTTIF and other cases

Figure 10 shows the effect on throughput for the three cases enumerated above. These tests were run using ttcp with a 200K socket buffer, and 8K writes. The overhead of VNET-VTTIF compared to VNET is a mere 4.1%.

5.5 Online VTTIF in action

Here we show results of running a parallel program in the online VNET-VTTIF system. We use the NAS Integer Sort (IS) benchmark for illustration because of its interesting communication pattern and traffic matrices. We executed NAS IS on 4 VMs interconnected with VNET-VTTIF. Here, the Virtuoso cluster, as used in the offline work, was employed. The rate-based reactive mechanism was used to intelligently trigger aggregation mechanisms on detecting traffic flow from the benchmark. When the benchmark finished executing, the traffic matrix was automatically aggregated at the VNET proxy. For comparison, we also executed the same benchmark with on 4 physical hosts and analyzed the traffic using the offline method.

Figures 11 and 12 show the topology and traffic matrix as inferred by the online system. Figure 13 shows the matrix inferred from the physical hosts using the offline method. The topology for the offline method is identical to that for the offline method and is not shown. There are some differences between the online and offline traffic matrices. This can be attributed to two factors. First, the byte count in VNET-VTTIF includes the size of the entire ethernet packet whereas in the offline method, only the TCP payload size is taken into account. Second, tcpdump, as used in the offline method, is configured to allow packet drops by the kernel packet filter. In the online method, VNET's packet filter is configured not to allow this. Hence, the offline method is seeing a random sampling of packets while the online method is seeing all of the packets.

The main point here is that the online method (VNET-VTTIF) can effectively infer the application topology and traffic matrix for a BSP parallel program running in a collection of VMs.

6 Conclusions and future work

We have demonstrated that it is feasible to infer the topology and traffic matrix of a bulk synchronous parallel application running in a virtual machine-based distributed computing environment by observing the network traffic each VM sends and receives.

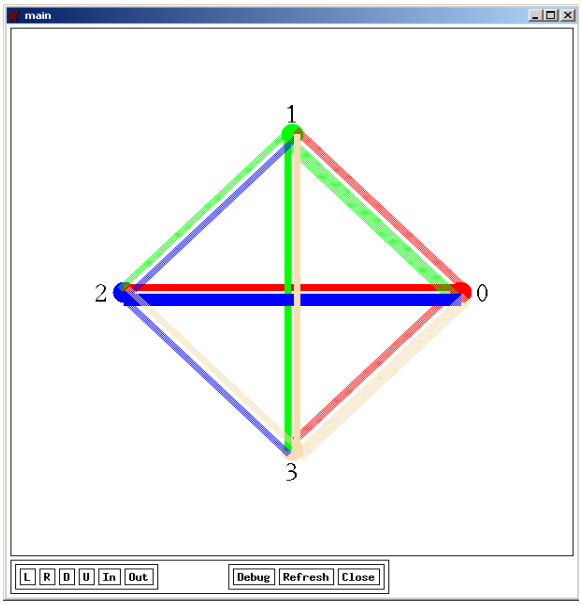

Fig. 11. The PVM IS benchmark running on 4 VM hosts as inferred by VNET-VTTIF

	h1	h2	h3	h4
h1		7.7	7.6	7.8
h2	13.1		6.6	6.5
h3	13.5	6.4		6.6
h4	13.2	6.5	6.5	
*numbers indicate MB of data transferred.				

Fig. 12. The PVM IS benchmark traffic matrix as inferred by VNET-VTTIF

We have also designed and implemented an online framework (VTTIF) for automated inference in such an environment. This monitoring can be piggy-backed, with very low overhead, on existing, necessary infrastructure that establishes and optimizes network connectivity for the VMs. We are now focusing on expanding this work in the following ways:

– We plan to generalize our results to other forms of applications and to determine the limits of network behavior that can be inferred.

	h1	h2	h3	h4
h1		5.1	5.0	5.0
h2	4.5		4.3	3.8
h3	4.7	3.9		3.8
h4	4.5	3.9	3.9	
*numbers indicate MB of data transferred.				

Fig. 13. The PVM IS benchmark traffic matrix running on physical hosts and inferred using the offline method.

- We are implementing a general query interface for querying traffic matrix information from our system.
- We plan to evaluate our system in more complex network environments, possibly revealing more filtering and topology inference based issues.
- We plan to improve the scalability and resilience of the system by adopting a distributed information aggregation approach.
- We intend to exploit the topological information provided by VNET-VTTIF to do optical call path setup on behalf of applications in networks that support it.
- We are working on leveraging VNET to do passive network measurement as a side effect of inter-VM data transfers.
- Finally, we are working on adaptation algorithms that will make use of VNET-VTTIF and network information to guide VM placement and migration, and VNET overlay topology construction and routing in order to maximize the performance of unmodified applications [9].

References

1. BAILEY, D. H., BARSZCZ, E., BARTON, J. T., BROWNING, D. S., CARTER, R. L., DAGUM, D., FATOOHI, R. A., FREDERICKSON, P. O., LASINSKI, T. A., SCHREIBER, R. S., SIMON, H. D., VENKATAKRISHNAN, V., AND WEERATUNGA, S. K. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications 5*, 3 (Fall 1991), 63–73.
2. DILGER, A., FLIERL, J., BEGG, L., GROVE, M., AND DISPOT, F. The PVM patch for POV-Ray. Available at http://pvmpov.sourceforge.net.
3. DINDA, P. A., GARCIA, B., AND LEUNG, K. S. The measured network traffic of compiler-parallelized programs. In *Proceedings of the 30th International Conference on Parallel Processing (ICPP 2001)* (September 2001), pp. 175–184.
4. FIGUEIREDO, R., DINDA, P. A., AND FORTES, J. A case for grid computing on virtual machines. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS 2003)* (May 2003).

5. GEIST, A., BEGUELIN, A., DONGARRA, J., WEICHENG, J., MANCHECK, R., AND SUNDERAM, V. *PVM: Parallel Virtual Machine*. MIT Press, Cambridge, Massachusetts, 1994.
6. GERBESSIOTIS, A. V., AND VALIANT, L. G. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing 22*, 2 (1994), 251–267.
7. LEIGHTON, F. T. *Introductio to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufman, 1992.
8. LOWEKAMP, B., AND BEGUELIN, A. ECO: Efficient collective operations for communication on heterogeneous networks. In *Proceedings of the International Parallel Processing Symposium (IPPS 1996)* (1996), pp. 399–405.
9. ROLIA, J., PRUYNE, J., ZHU, X., AND ARLITT, M. Grids for Enterprise Applications. In *Proceedings of the 9th Workshops on Job Scheduling Strategies for Parallel Processing (JSSPS 2003)* (June 2003).
10. SAVAGE, S., COLLINS, A., HOFFMAN, E., SNELL, J., AND ANDERSON, T. E. The end-to-end effects of internet path selection. In *SIGCOMM* (1999), pp. 289–299.
11. STASKO, J. Samba Algorithm Animation System. Available at http://www. cc. gatech,edu/gvu/ softviz/algoanim/samba, html.
12. SUNDARARAJ, A., AND DINDA, P. Towards virtual networks for virtual machine grid computing. In *Proceedings of the 3rd USENIX Virtual Machine Research And Technology Symposium (VM 2004)* (May 2004), pp. 177–190.
13. WHITE, S., ALUND, A., AND SUNDERAM, V. S. Performance of the NAS parallel benchmarks on PVM-Based networks. *Journal of Parallel and Distributed Computing 26*, 1 (1995), 61–71.
14. ZHANG, Y., DU, N., PAXSON, E., AND SHENKER, S. The constancy of internet path properties. In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop* (May 2001).