# Taking Advantage of Multi-cores for the Lustre Gemini LND Driver

James Simmons
Oak Ridge Leadership Computing Facility
Oak Ridge National Laboratory
Oak Ridge, TN 37831, USA
e-mail: simmonjs@ornl.gov

John Lewis
Customer Service
Cray Inc.
Seattle, WA 98164, USA
e-mail: lewisj@cray.com

***Abstract*— High performance computing systems have for some time embraced the move to multi-core processors, but parts of the operating system stack have only recently been optimized for this scenario. The Lustre file system has improved its performance on high core-count systems by keeping related work on set of cores that share a NUMA node or cache, though low-level network drivers must be adapted to the new API. The multi-threaded Lustre network driver (LND) for the Cray Gemini high-speed network improved performance over its single-threaded implementation, but did not employ the benefits of the new API. In this paper, we describe the advantages of the new API and how the Gemini LND performance was impacted. We will take a detail look at various setups to determine what is the best possible configuration.**

***Keywords: Gemini, Lustre, multi-cores, SMP***

## I. Introduction and Motivation

World wide HPC centers have enabled scientist to resolve complex problems. With these solutions even larger questions about the nature of the problem come to light. To handle these very increasing demands HPC centers are at the forefront of expanding technological horizons. In 2012 ORNL, to meet the increasing needs of its users, upgraded the main computational resource Jaguar to a new architecture and renamed the machine to Titan. The upgrade changed the machine from an Cray XT5 machine using an SeaStar interconnect to the Cray XK7 platform which has the newer Gemini interconnect. Part of the process of these upgrades is to evaluate what the benefits and limitations are for the new hardware. These analysis help us to determine such things as optimal application placement and best router configuration to maximize the through put to the file system. The expansion of the machine has placed additional pressures on the current file system which will be remedy with the deployment of the next file system.

Internal data on the Gemini interconnect have shown a peak of 7 GB/s between compute nodes at packets sizes of 64KB and up. When transmitting 4KB size packets a sustained output of slightly above 3GB/s is observed. This is the ideal conditions so some loss is expected when this network layer is implemented in different software stacks for specific use cases. The question becomes how efficient is the software stack to minimize the loss experience and if the loss is substantial what can be done to improve the performance.

The area of impact that this paper explores is to the parallel file system deployed. The file system of choice at ORNL for production is the Lustre file system. Lustre possess a abstraction layer called LNET to enable supporting many network topologies in a uniform way. To take advantage of this abstract for the Gemini interconnect an LND driver had to written to interface with the LNET layer. As stated before some loss was expected off the raw values but our studies showed how poorly utilized the bandwidth, currently 1.6GB/s, was on the Gemini LND. Data collected pointed to a large penalty from the checksumming that was performed on data packets being processed by the LND driver. The problem expressed itself in two ways. The first was the check sum algorithm itself performed poorly with large data sets. Secondly the LND driver did not scale to handling increased network traffic coming in. In later versions of the driver's code more threads were added to help with performance but we still saw significant penalties with having checksums enabled. Disabling checksumming seems to be the logical choice but one risk allowing potential corrupted data being written to disk. Secondly even with checksumming disabled the maximum performance observed was 4 GB/s between a compute node and service node.

To meet these challenges the research of this paper explores leveraging modern hardware to decrease the losses experienced with the Gemini LND driver. First we will examine the current behavior of the LND driver to evaluate were the bottle necks occur. This paper details the various combinations tested to validate the theoretical configurations that would give the best performance. Using this data we show how to approach future hardware platforms to best utilize the hardware. Also we go over the future direction of the driver's software stack to over come the limitation of the checksumming algorithm itself.

## II. Theory

The current challenges to the Gemini LND driver were also problems for the Lustre file system in the past. Today we can adopt their solutions but first lets look at the history of the problem for Lustre and how it was solved. One of major bottle necks for Lustre is the metadata server not being able to scale. The problem can be attacked in two ways, one being to spread the meta data across multiple servers which is being developed in the new DNE frame work that is partial completed. The other approach was to look at vertical scaling by taking advantage of multiple core machines that

the MDS resides on. The SMP scaling work that came out of this research was also applied to other layers of the Lustre stack to increase performance. Lustre version 2.3 and above ships with the SMP improvement completed and integrated into the stack.

Since the focus of this paper is improving a LND driver let us study the problems that hindered the LNET that were resolved by the SMP changes. As larger file systems were deployed the limitation was not the disk but the fact that the back end servers would easily become CPU bounded which resulted in the Lustre clients ending up in a soft locked condition. Their were a few reasons for what triggered these conditions. Lockmeter analysis showed that large contention existed for the global locks that were present in the LNET layer at that time. The source of contention was the lock being synchronization so a consistent view of its state was presented to every core. Since memory has a lower clock rate then the processor large latency would occur. Other data types, such as the wait queues, in the stack also were impacted by this contention. In the case of wait queue an additional penalty would results from the natural round robin nature of handling the next event. By the time we came back to the same thread in the queue the data would have to be migrated into an CPU's cache. Besides the penalty of cache migration the data being migrated into the cache's rarely were laid out to minimize cache line misses. This problem is enabled by the lack of CPU affinity for the kernel threads.

With a understanding of the problems the SMP api was developed to address these issues. The center of the design is the CPU partition which is a grouping of cores. The premise of the grouping is to place together cores that share a cache or belongs to the same NUMA node to minimize the memory penalty. Default CPU partitions are created based on how the Linux kernel sees the topology of system. Usually this is the best case layout but we shall see this is not always the case. To take advantage of the memory locality one needs to use the per-partition and partition local memory allocator routines in your code. With the localized memory one can avoid the penalty of the thread migration when the thread is bind to the proper CPT. One needs to ensure the proper cores belong to the same CPT so that the thread migration between cores in the CPT has no penalty. Having parallel handling of events with multiple threads lacking contention enables far better scalability.

With any well written api you give your users choices in system behavior. Here we go over the setting that have a impact on LNET layer as well Gemini LND driver. The center of the api is the CPT so we shall go over that first. To get the CPT (CPU partition table) the command *lctl get_param cpu_partition_table* is used. In the example output below we get a visual of the layout of cores.

```
cpu_partition_table =
    0  : 0 1 2
    1  : 3 4 5
```

In this example we have two partition tables with table one containing cores 0, 1, and 2. This layout can be controlled with two types of module parameters for the libcfs module. The first approach is to control the number of CPTs created. An example of this would be

```
options libcfs cpu_npartitions=3
```

In the above example we would have three unique partitions with each partition having a total of two cores. Care must be done with the number of cores in the partition. Any core has the potential of becoming CPU bounded so having a CPT containing only one core would prevent the LND thread from migrating to a non blocking core. Sometimes one needs even finer grain control of the layout so a method is provided to control which cores belong to a CPT. Using the same example we change the cores of the CPT to contain each only the even number or the odd number cores.

```
options libcfs cpu_pattern="0[1,3,5] 1[0,2,4]"
```

If you have many partition you can use short hand notations such as 0[0-20/2]. How this is done depends on the architecture of your machine. In the case of Cray hardware this will be handled differently on the service nodes versus the compute nodes. Service nodes and compute nodes contain not only different number of processors but these processors can also be different types. The api is flexible enough to require you don't need to use all the cores on a system. Let us take the case of limiting the OS noise on a compute node to limit the impact to an application. In this case the job is launched with aprun -r to have all the OS specific handling to be migrated to a specific core. The stock Gemini LND driver doesn't follow this behavior but carving out a single CPT containing two cores would remedy the situation. Having two core would prevent the IRQs or IO handling from stalling one another. This would further optimize lowering the noise level on the compute nodes. One does need to take note which NUMA node the Gemini interconnect is attached to with the Hyper-Transport bus so the CPT belonging to the same NUMA node is used for performance reasons.

The CPT you create will be the ones used by the LNET layer. The impact this has on the LNET layer was the ability to have localized buffers and have thread pools bounded to specific CPT. Parallel event queues handling incoming traffic enables for far better scalability plus you receive less lock contention. To maximize this advantage the LND driver needs to aware of the LNET CPT that come into play. Like the CPT being configurable the administrator can control which CPT subset each LNET interface can be aware of. Using a 6 core 2 CPT example lets say you want the Gemini LND driver to only use the second partition. For this example the module parameter option would be

```
options lnet networks="gni0[1]"
```

Remember having your LND driver use less CPT than the LNET layer can impact performance since data can migrate from one CPT in use by the ptlrpc thread to the CPT in use by your LND driver. Usually this case is used in situation that you have more than one physical network interface.

Now that we understand the components behind SMP scaling the next step was to apply it to the Gemini LND driver. To properly implement the api we have to lay out the

proper mapping between the variable number of components. For the typical LND driver we have:

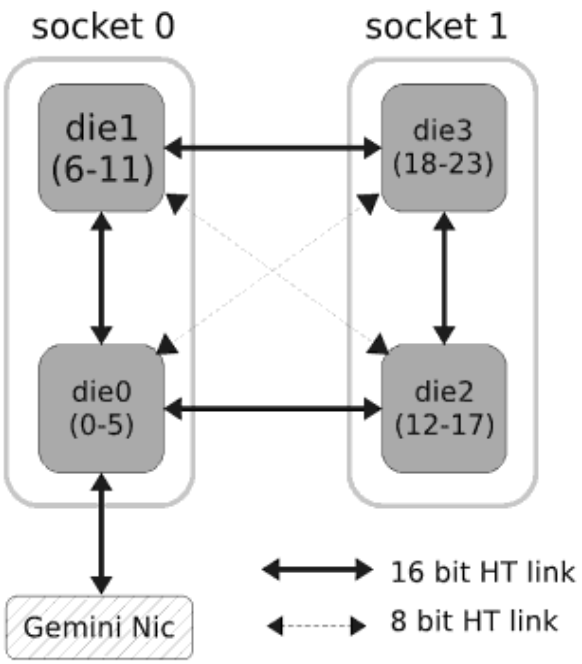
X LNET interfaces : Y devices : Z CPT

Most drivers only use one LNET interface per device but this is not the case for our driver. On Cray systems the LNET layer is used by Lustre and DVS. Each uses the same gni interface but are registered to different Portal ids. In reality for Lustre the LNET configuration of a Gemini driver really looks like 12345-30@gni0. You can see this result when you do a lctl ping. In the case of DVS the port will be some thing besides 12345. So in this case we will have 2 LNET interfaces. Currently the hardware only support one device but the driver is written to take advantage of multiple devices if a platform ever comes into existence. Lastly we have to handle mapping the CPT to each device. Nothing stops devices from having over lapping CPT. In our case of DVS and Lustre we want to share the CPT setting for each LNET interface so as not to degrade performance.

## III. Test equipment architecture and configuration

In the previous section we discussed the foundation of this work in order to understand how to apply it the specific machine architecture. How each machine is configured is hardware dependent but the rules for optimal layout can be generalized such that it is easy to apply the same principles to a new machine and get the improvements we expect. For our test system we have a single cage Cray XE6 which we named Arthur. In our test system configurations we have total 20 compute nodes, 6 router service nodes, and 2 login service nodes in use by the file system that is supplied by a DDN S2A9900.

This platform has no GPUs and the CPU installed are AMD Magny-Cours. In total 6 cores are placed within a NUMA region (or die) sharing a 6MB L3 cache with each core having a 512KB L2 data cache and 64KB L1 data cache. For compute nodes as we can see below we have two sockets were each socket contains two dies with 6 cores embedded each thus a total of 24 cores per compute node.




The service nodes have a single socket which contains a total of 6 cores. For both computes and service nodes the Gemini interconnect communicates directly with the cores on NUMA node 0 using the Hyper Transport bus. As we can see in the figure which represents the compute node case, we want to divide up the system in four different regions. Using the wrong NUMA region for the Gemini NIC will come with a performance penalty. This is a case were we want to avoid using any cores not on NUMA node 0. In testing we will explore if we can saturate the Gemini interconnect using only cores on NUMA node 0 and if we need to use cores on other dies study what that cost is.

With the ability to use more than one core on a NUMA node introduces with it the potential cost of impacting an running application. So the question is how does one minimize this. On our Cray XE Magny-Cours test system each processor has its own FPU so there is no easy solution to this problem. In the future out work will be migrated to Titan which uses the AMD Interlogos chipset. Separate experimentation showed using Linpack that on Cray AMD Interlogos based systems you could get nearly the same performance using only half of the cores on node if you correctly picked the cores to run on. The reason for this is that on AMD Interlogos processors every two cores share a FPU. Taking advantage that most scientific applications only care about using the FPU ideally we want at most 4 well placed threads on the compute nodes. This can take place because the Gemini LND driver will never touch the FPU. Normally the times when a application doesn't use the FPU are when it has to deal transferring data to the file system which in our setup will be taken advantage of.

## IV. Test cases

### A. *LNET and LND configurations*

Now that we have a good grasp of the hardware we can define what our test parameters will be. With the service nodes having only one socket with a single NUMA node containing six cores so the combinations are limited. For the stock Gemini LND driver 3 threads are created to handle the traffic that is caused by the high ratio of compute nodes to individual routers. Those threads do not have any affinity with any cores so they are free to migrate. Our current test system only has twenty compute nodes but increasing the kgnilnd threads on the compute can help create an environment that simulates having a larger set of compute nodes. We can use this setup to explore effectively what the impact of more threads on the service node will have on performance. Thus the other option is running 6 threads to see how if it has any influence on performance.

Moving to a 2.4 version of Lustre we see by default libcfs creates two CPTs on the service nodes with each CPT containing 3 cores. If we consider that the L3 cache is shared with all cores the only advantage of having more than one partition is to increase parallelism of the handling the events. As with all CPT configurations having one CPT with all cores or having too many CPTs are costly so those test scenarios can be omitted. This leaves us with testing CPTs

containing two or three cores to evaluate its interactions with the largest set of compute nodes we have in our test system. Experimentation will determine which sets of cores are the optimal setup.

For the case of compute nodes the stock Gemini LND driver takes a conservative approach by using only one thread to handle all traffic. Without CPU affinity this thread is free to migrate to any one of the 24 cores on the node. Earlier in this paper we discussed the reasons why the cores on NUMA node 0 are optimal. In the case of the default driver we see that the single thread is allowed to freely migrate to another NUMA node which incurs an extra penalty. Since only one thread is used the question is whether adding more threads add value. This question will be answered by increasing the number of threads up to the 24 cores present in the system. To make the comparison for both SMP and non SMP cases the number of cores for each set of tests will be identical.

Compute nodes that have Lustre 2.4 clients installed for our hardware platform are automatically setup with 4 CPTs each containing 6 cores. By default to match the original behavior of the stock LND driver we also only start a single thread on each node. For the case of using less than six threads all spawned threads will be located in the same CPT. While our LND driver will be using one CPT the LNET layer will be handling data on all the CPTs. A penalty will be encountered when that data has to be transferred to a different CPT. The initial set of test will focus on creating CPTs only for NUMA node 0 to minimize the path to the Gemini interconnect. Much like the service nodes the options for CPT configurations on each NUMA node is limited. Using the data collected from the service node runs we will have a grasp on whether a NUMA node using CPTs containing two core or three cores is the better choice. In either case we will have the second layer of testing using the cores on NUMA node one and then using both NUMA nodes on socket 0. The last setup described is the most optimal relative to applications but to make the test complete we will examine the case of using all cores on socket 0 as well as having six core in use on socket 0 as well as six more cores on socket 1. Even though we have the same number of cores this will show the impact of placement will have. The final set of test will use all the cores on the compute node. At some time during testing we will witness the saturation of the Gemini interface. At that point no extra amount of threads or placement will have an impact.

### B. *Benchmarking software*

Our testing will be modeled after the original SMP work done under the OpenSFS contract with Whamcloud. Based on that work we see the focus of their testing was done with LNET selftest and mdtest. The reason for the lack of bulk message study is due to the interaction of the bulk messages with the cache. The larger the data packet the more probable that some of that data will be flush from the cache especially as time progresses. LNET selftest is a tool distributed with Lustre can be used to analysis the performance of the LNET layer itself. In the course of our studies we will do a impact study to ensure that the SMP scaling work does not degrade the current bulk message handling. Due to time constraints mdtest were not performed.

With LNET Selftest you can create batches of test which have several controlling factors. The basic LNET functionality that is exercised are LNET ping rates, and read write transfer rates. In the case of reading and writing different sizes messages can be sent. The most common message size observed on the production file system at ORNL are 4K and 1M in size. These sizes will be the focus of our testing. Also both these sizes will reveal the impact the SMP changes to both small as well as bulk messages. Along side these test parameters we can also control how many concurrent request are outstanding. For most test conditions we will have the number of outstanding request to equal to the number of cores on our compute nodes to ensure that each node is saturated. One special set of test do exist that have one node communicate with a another node with every increasing number of request to see where the upper bound exist for sending request. This will have a influence on the proper number of threads to spawn on each node as to not waste cores which could be used for other tasks. Besides selecting which basic functionality these tests can be done with different combinations of senders and receivers. In one set of tests we will be looking at one to one communications. In this test group we will look at compute to compute, compute to service, and finally service to service node interaction. The other set of test will be many computes nodes to one service node which is the typical production communication pattern. Here we want to study how well the routers handle a scale up in the number of clients communication simultaneously with increasing traffic.

## V. Results

### A. *Smoke test*

One of most basic test done to evaluate the performance of the network fabric is the smoke test using LNET selftest. This test floods the network with synthetic traffic with all nodes communicating with each other. This is the test that can most saturate the network. For our test bed this was done with 16 compute nodes and 4 routers send network traffic to each other.

TABLE I. LNET SMOKE TEST RESULTS

| Compute node thread count | Service node thread count | CPT count | 1MB writes transfer rate |
|---|---|---|---|
| 1 | 3 | unimplemented | 2227.19 MB/s |
| 1 | 6 | unimplemented | 2272.32 MB/2 |
| 1 | 3 | 2 | 3625.74 MB/s |
| 1 | 6 | 2 | 3650.57 MB/s |
| 1 | 6 | 3 | 3180.05 MB/s |
| 3 | 6 | 2 | 3672.71 MB/s |
| 6 | 6 | 2 | 3692.82 MB/s |
| 12 | 6 | 2 | 3711.70 MB/s |
| 24 | 6 | 2 | 3689.49 MB/s |

First lets look at how different configurations on the service node impacts the smoke test. From this set of data one can see that for the default driver adding three additional threads on the service nodes increases the amount of network traffic being handled. When you compare both cases of a different thread count on the SMP enhanced driver we see negligible improvements at this ratio of compute nodes to an router. This was the case when our compute nodes were running with a large number of kernel threads as well. So most likely for many compute nodes the addition of threads will make no difference. When one compares the SMP to the non SMP case you see a definite improvement.

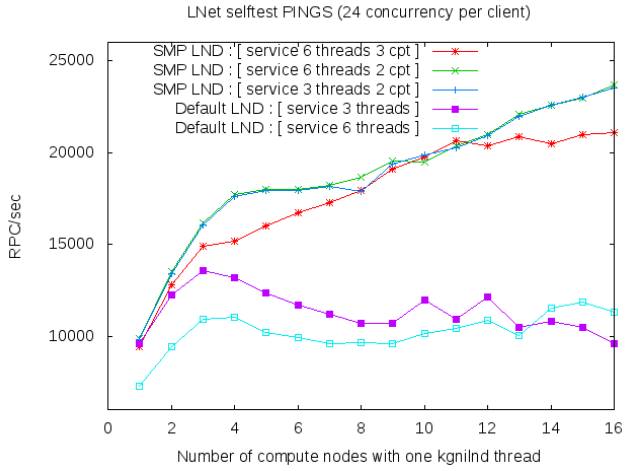
### *B. Increasing compute node to router ratio*

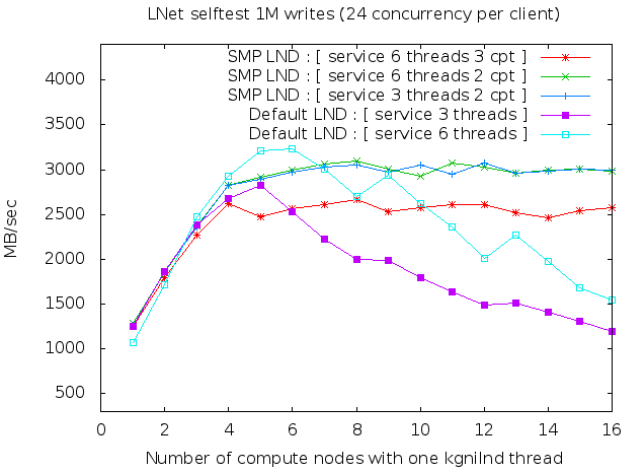
While the smoke test is a interesting test case what is critical to installations is how well a router will handle the network traffic coming in from many compute nodes. The routers can easily become the bottlenecks in the file systems performance. The environment is similar as the smoke test in that we use 16 compute nodes communicating with one router. Like the smoke test we will vary the number of threads on both the compute and service nodes. This analysis will need to broken to two different parts for clarity due to variability that can be done on the compute nodes as well as the server nodes.

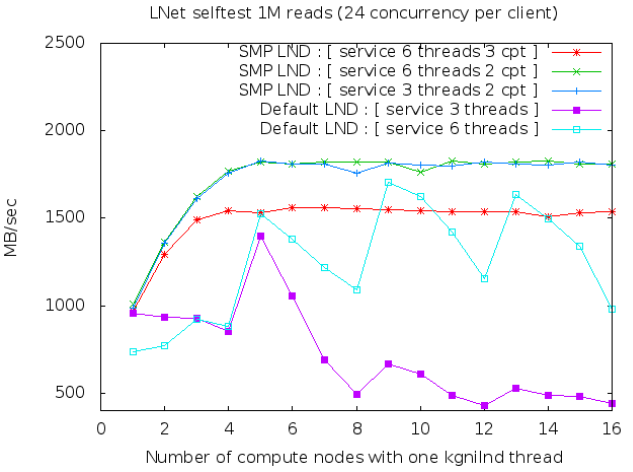
#### *1) Service node configuration effects*

First we will examine the case were the compute nodes will be using the default one thread in the LND driver. The variability will be solely on the service side to discover which combination is the best for the service node. As detailed early since the service node has less combinations to experiment with we can eliminate the less optimal conditions. This will help with limiting the number of test for the compute node. In all case we will examine the different test scenario covering 1M and 4K read as well as writes of the same size. LNET ping also will be included to show the network behavior without the impact of check summing.

Inspection of the LNET ping test results reveal for the default stock LND driver only after a few compute nodes we reach a saturation point and remain at approximately that level independent of the number of compute nodes pinging the router. For the SMP enabled driver a linear scaling is observed for all test configurations. By default Lustre 2.4 SMP layer creates two CPTs but in one of the set of test three CPTs are created. Intuitively you would think that have three parallel threads handling the network traffic would give better performance but this is not the case. What is observed is each thread is experiencing greater competition for access to an core since it now has fewer less busy cores to migrate too. Since this is a many computes to router test we can't say that the router node is being not driven hard enough. When we look at the SMP cases with identical CPT counts but different threads being created the data reveals nearly identical behavior. With a small number of computes we have demonstrated that have 6 threads added only a small margin of gain. At the same time it also shows us that extra threads are not a penalty either. This is some what expected since all the service node cores share the same L3 cache.

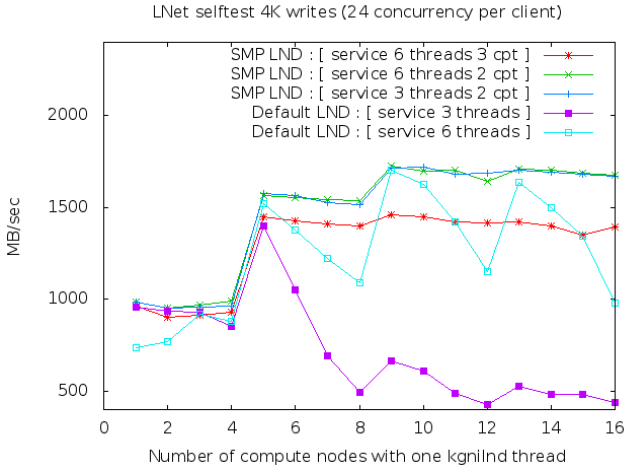


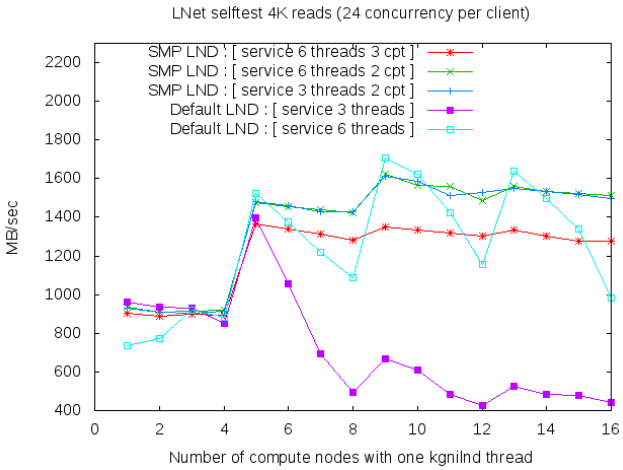




When we look at the data for the read and write cases we see the impact of the check summing has on scaling. For the SMP case we see a ramp up to five nodes and then a flatting

out of the amount of data we can handling. This is for the 4K and 1MB size cases with the only difference is that for small packet sizes we see some thing similar to a step function when we reach the fifth node. Remember this data represents how the service node is handling the incoming or out going data. The thought here is that the sum of the amount of data being received grows in scale with the number of computes interacting with the router. As this amount of data grows then those larger memory chunks can be handled more efficiently. As you will note that again creating three CPTs instead of two causes a performance hit. Also with the SMP work the we experience nearly constant handling of network data where as the default driver displays greatly varying performance.

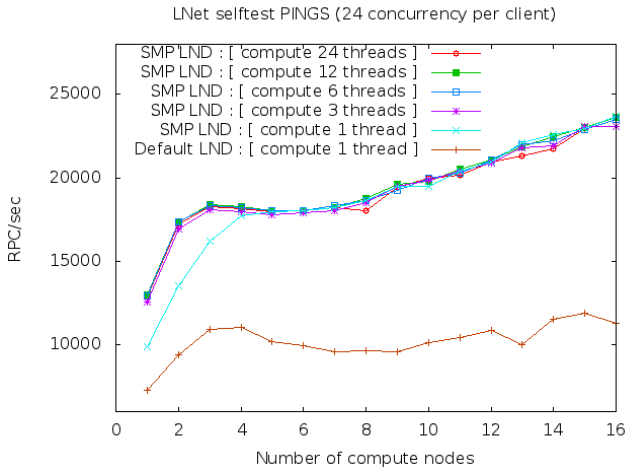



We can conclude from the data collected that the SMP work has helped not only improve performance but also made that performance consistent at larger scales. We see that more CPTs does not guarantee greater returns but at the same time adding more threads come at very little cost on the service node. The best results here will be used as the control group for the other test sets that follow. For the case of deployment the best setup on a router would be one CPT with three threads for the Gemini interconnect and one CPT with three threads for the infiniband interface.

*2) Compute node optimization*

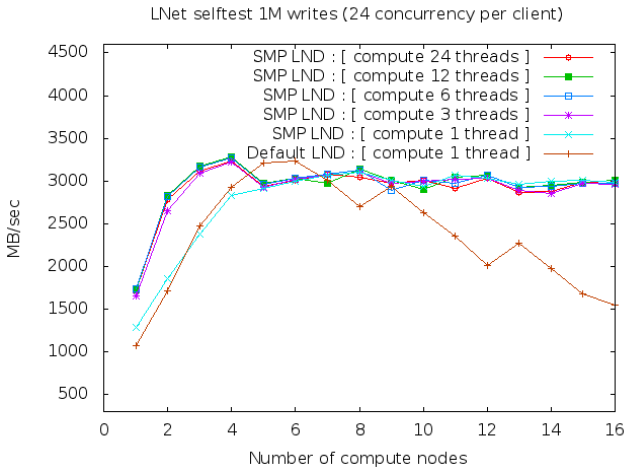
For the following set of tests we will keep the service nodes in the same setup but vary the compute nodes instead. For the service nodes we will use a two CPT six thread setup since this was the best setup for the default stock Gemini LND driver. This ensures that the same test parameters are uses for the patched and non patched LND driver cases. We will use the default CPT setup for the compute nodes since they map nicely to the hardware topology. By default 4 CPTs are created with each CPT mapping to the six core die. Here we will analysis how the service node handles the traffic changes in behavior due to increasing kernel threads on the compute nodes. In this section we will focus mainly on the different SMP configurations. The reason for this is the non SMP case showed a collapse in performance when increasing the number of kgnilnd thread when data was collected for the node to node test. From that we know that one thread on a compute node with high concurrency has the best performance.

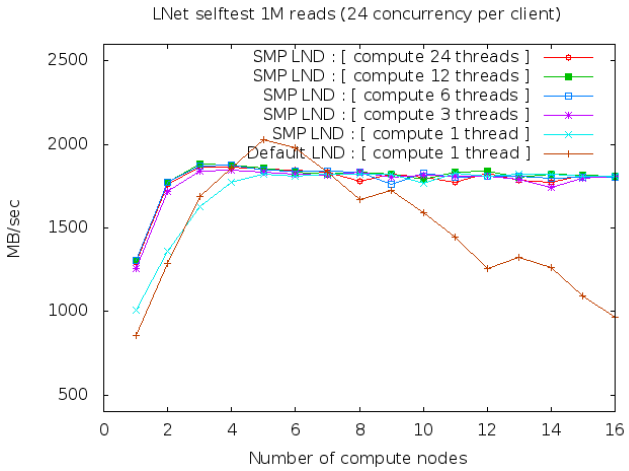



We can see that ping scales very much like the service node case. Also like the service node case we see the default LND driver top out at about four compute nodes and remain at the level. As you can see a very large performance gain occurs. What is interesting is we see nearly identical performance with five or more compute nodes independent of the number of threads. This shows too many extra threads has no advantage at all. In the six thread case we do see improvements when dealing with less than five compute nodes.

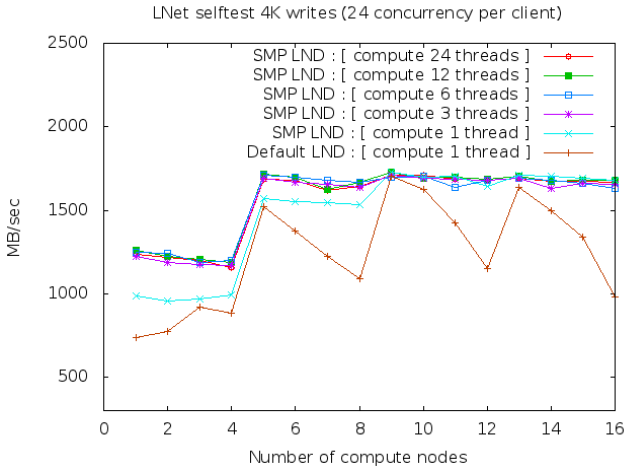
Looking at the results for reads and writes measuring 1MB in size we see similarity in behavior. For the default LND driver scaling occurs until we reach the fifth compute node communicating with the router. After this point the amount of data being handled decreases. When you compare it to the SMP cases we reach saturation earlier than the default case and then a leveling off in the amount of data being processed. As you can see no decrease in performance

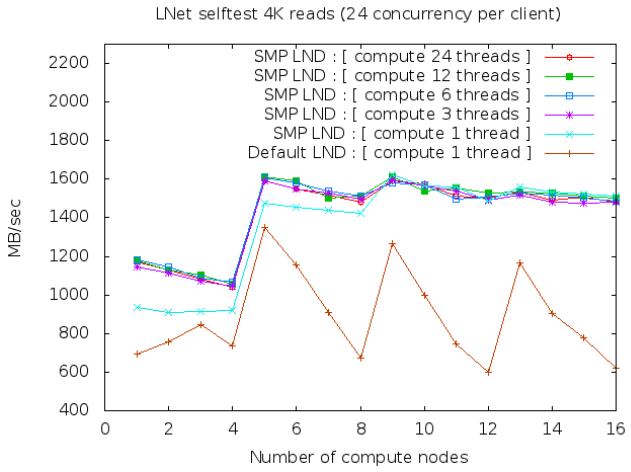
occurs in the SMP case. Only in the read case does the default driver outperform the SMP driver in one special case. As we observed from the ping results when the computes go from one thread to three a performance gain is experienced when a small number of compute nodes are communicating with the router. Future testing with multiple routers will determine if this is router saturation.




When dealing with 4K reads and writes we see a less smooth behavior for the default driver case. As with the 1MB results we see a peaking in performance around five compute nodes to one router. Instead of a steady decreasing performance the data transfer rates become range bound oscillating between the envelope defining its range. If you look at the earlier figures for the service node you see a very similar behavior for 4K packets. If we were to test with only three threads on the service node the performance would not only oscillate but continue to decrease as we would scale more compute nodes. The six threads for the default case does not experience this decay. Once you turn your attention to the SMP results you will that the oscillation is absent. The performance gains in this case were also present. Again the gains added are negligible when adding more threads on the compute nodes. The reason for no gain with the twelve threads and above is that we incur a penalty accessing the cores located in a different NUMA region. This penalty erases all possible gains. When dealing with a smaller pool of compute nodes we see that three threads on the compute node instead on one gives a noticeable gain.



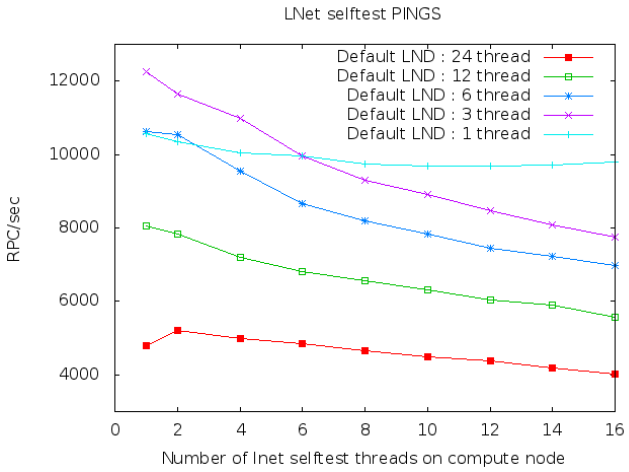

### *C. Node to node behavior*

So far we have looked at the over all behavior of the system and with the SMP work witnessed improvements. The metrics we have so far gathered are defined by the aggregate of all the nodes that compose the network. In this section we will look at node to node behavior to see the lowest level alterations in performance. Due to the nature of the Gemini hardware this set of test will not reveal the highest amounts of traffic since we will not be transversing all the directions of the torus. Node to node evaluation does gives us an insight in how the LND driver behaves when varying levels of data are being pushed through the LNET layer's software stack. To simulate this both nodes that will be communicating with each other will be loaded with an increasing number of LNET self test threads being created to push data on the network. Three cases of node to node interaction will be examined. The first is compute node to

compute node, second one compute node to a service node, and the final case is service node to service node. For cases involving the compute node two separate graphs will be displayed. One for the SMP case and the other for the default LND driver. For both cases we will examine 1, 3, 6, 12, 24 kgnilnd threads being spawned on the compute node.

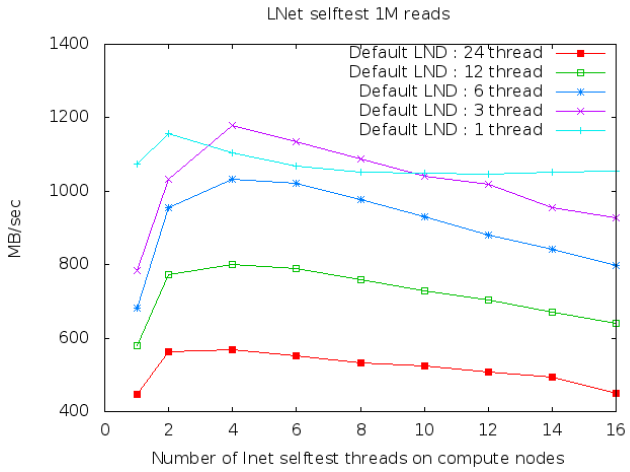
### 1) *Compute node to Compute node*

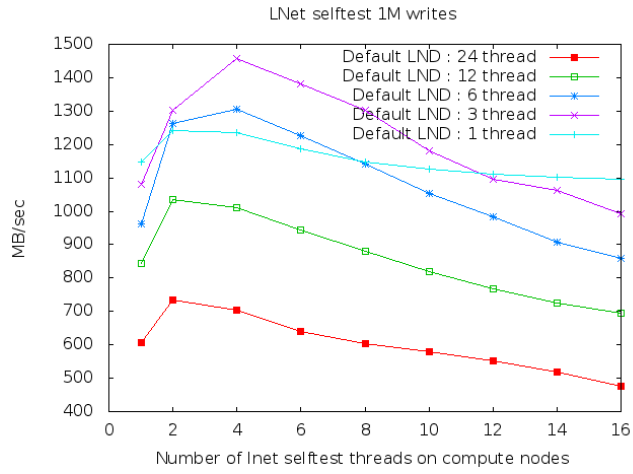
From previous test it was demonstrated that using six or more threads on the compute node add very little gain. For the case of one thread verses three threads the gain was only observed for a small pool of compute nodes when using the larger number of threads. The question is it worth that small increase. To get the proper answer to this question properly examining compute to compute node behavior is required. Before we look for this answer we need to examine the default drivers characteristics when adding these additional threads.

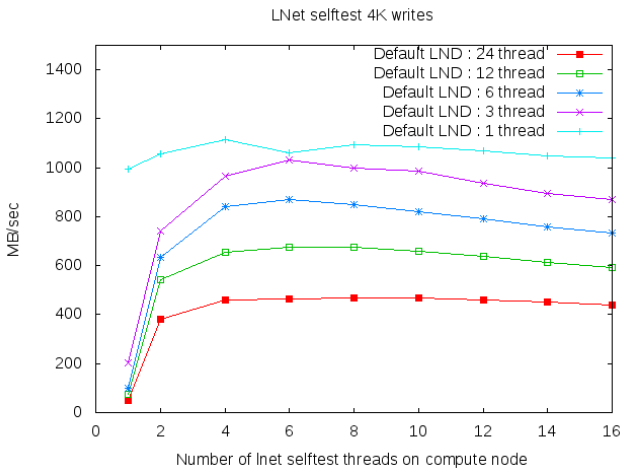



For the default LND driver each time we increase the kgnilnd thread count the performance degrades even more. The only except happens when we go to three kgnilnd threads but those improvements are conditional. As LNET self test pushes its concurrency above five the three thread case begins to degrade at a much faster rate then the single thread setting. In all cases including the default single thread case as the concurrency increases the performance drops off.

When we examine the read and write cases the same performance is observed. Between the 1MB and 4K results a similar pattern emerges. In the case of the 1MB tesst case the single thread case has a small peak plateau around two to four nodes with a mild decline with increasing concurrency for LNET self test. At three threads a small gain occurs but with increasing concurrency contention eliminates all gains. While we might get some benefit for three threads with bulk messages when one looks at the 4K size messages we see a total lose for any increase number of threads on the compute node. For the non SMP case we are seeing the impact of thread migration especially as well increase the number of threads. More threads means more flushing of the cache when the thread is migrated the next time it is scheduled to run. This would only be more amplified if a intensive application was running across these compute nodes.

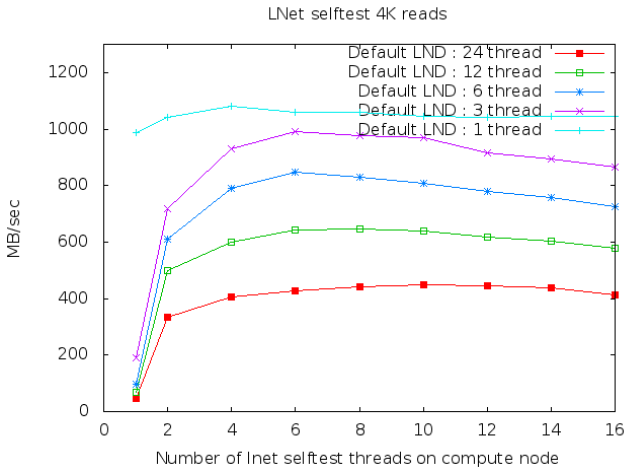


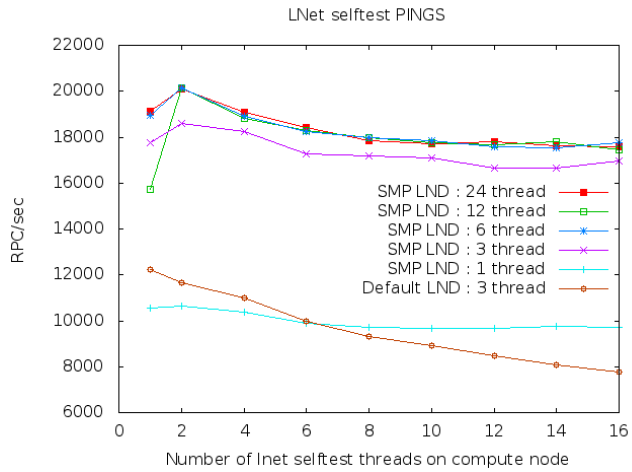




From these results and the 4K read that follows we see that the best case scenario is the one thread or three three case depending on which functionality you are testing. Since checksumming can influence the results the data for the

LNET pings gives us the clearest picture of how adding more threads impacts the performance. If we weigh the ping results most heavily then the three thread for the default driver will be used as are baseline against the SMP results.

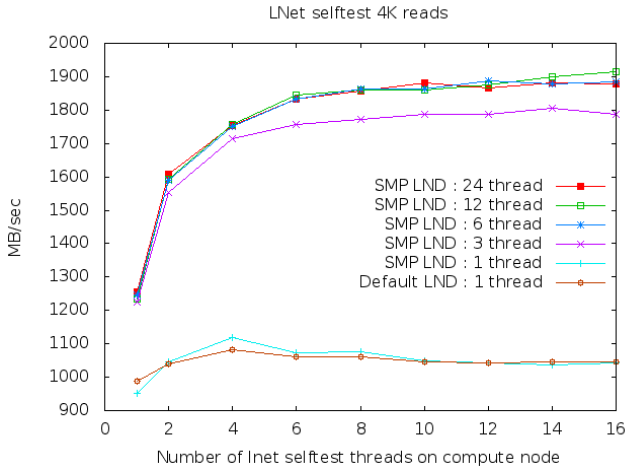


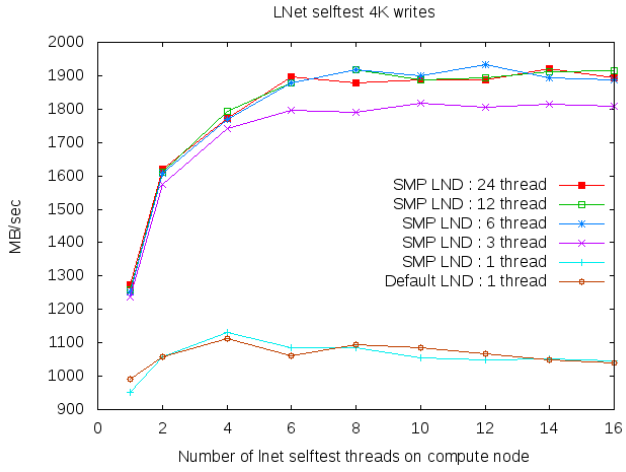

The next set of results are for the runs using the SMP patched driver. The same set of thread counts will be used as the non patched driver version. The SMP results will be compared with the best results of the default driver. From the results in the last section we see that peak performance occurred when the thread count was three on the compute node. Looking at the below graph The first thing that stands out is the reversal of the impact that adding more threads has. Each increase in thread count comes with gain instead of a loss. In the SMP case going from one to three threads has enormous gain. Each increase in thread count after three threads comes with no penalty as well as no gain. From this we can conclude more than six or more threads add no benefit.

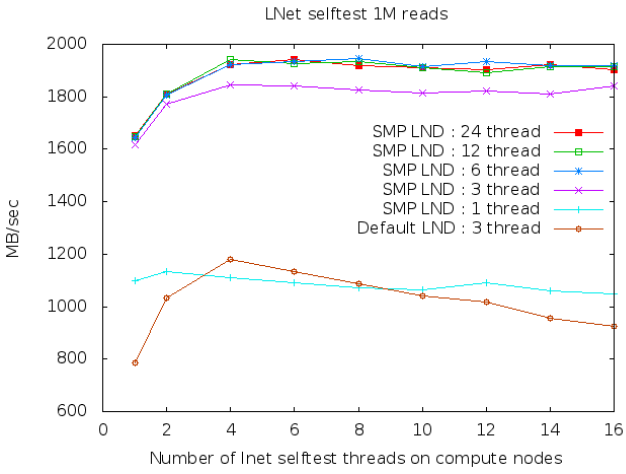


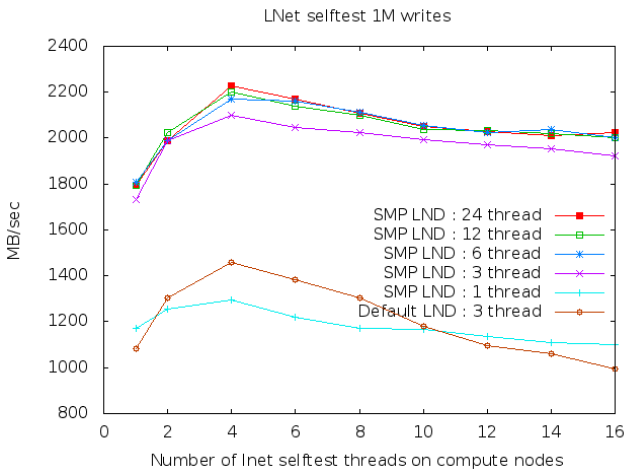

We do see a small peak around two lnet self test then a small dip with a leveling off. With the SMP work the contention that existed in the original driver has been removed. When comparing the both one thread cases for the SMP scaling work and the default driver you see a nearly the same behavior. We should observed this same type of behavior for the two classes of read and write test.

Our testing for 4K read and write for the default driver revealed that unlike bulk messages that the best performance was with the default one thread. So for this test case we will be comparing the SMP results to the one thread results of the default driver.



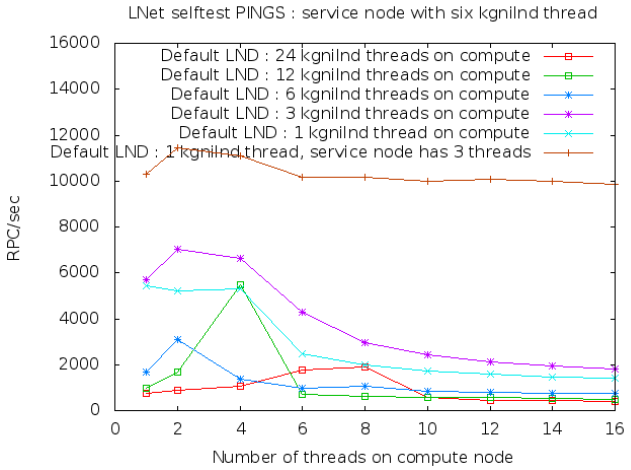



For both 4K and 1MB message sizes we see the exact same behavior which is not the case for default driver. Much like the service nodes behavior having to many threads adds no extra benefit. While it was not clear in the N to 1 test, or the N to M test we can clearly see that around three threads get the best performance for compute nodes.

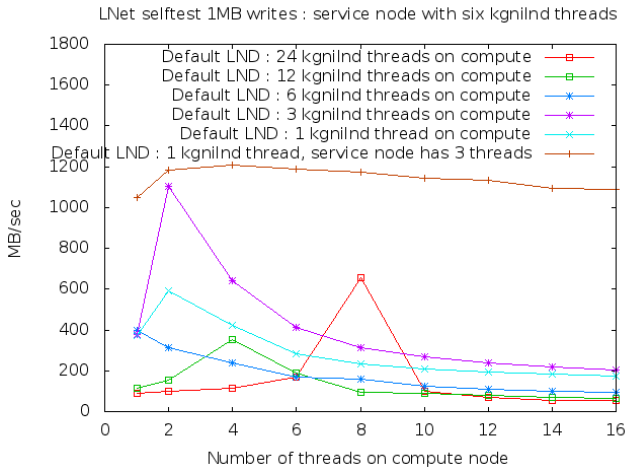
*2) Compute node to Service node*

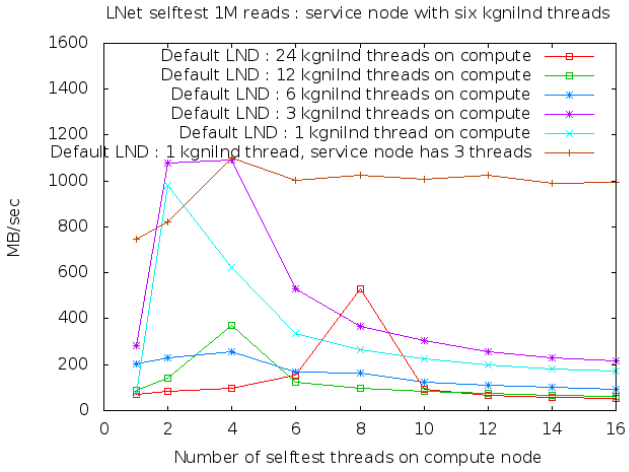
Earlier we did a detail study of many compute nodes communicating with one router. In this section we will look at the characteristics of a single compute node to an router. Instead of scaling more compute nodes we will, like the compute to compute case, scale the number of LNET self test threads being spawn. The data is sampled on the service node to see the affects of having extra kernel threads on compute nodes. Due to the large number of test cases we will look at the default LND driver and then separately study the SMP enabled LND driver.

So far the common behavior for default LND driver for this class of test is to peak quickly then follow a path of degradation. A show of the performance of the ping eliminates the impact of check summing. As expected the best configuration is using only one thread on the compute node with three threads on the service node. Increasing the number of threads on the service node nearly cuts the performance in half.

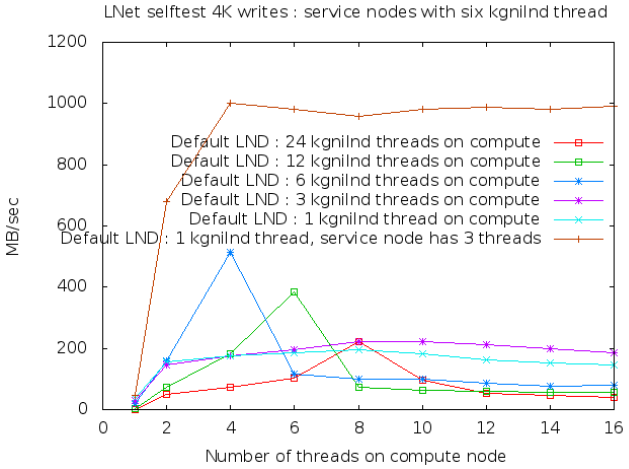


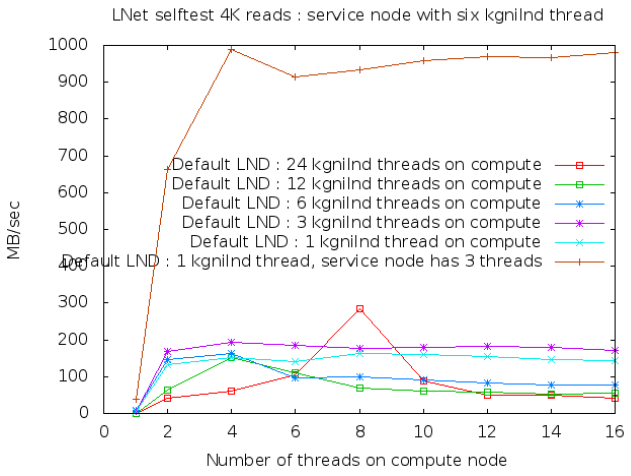

Continuing to the scale the number of threads serves to only degrade the system. We see the typical spike in performance and then a gradual decline. As the number of LNET self test increase on the compute node the more the network data rates converge together on the service node. This is true for all read and write conditions as seen below.



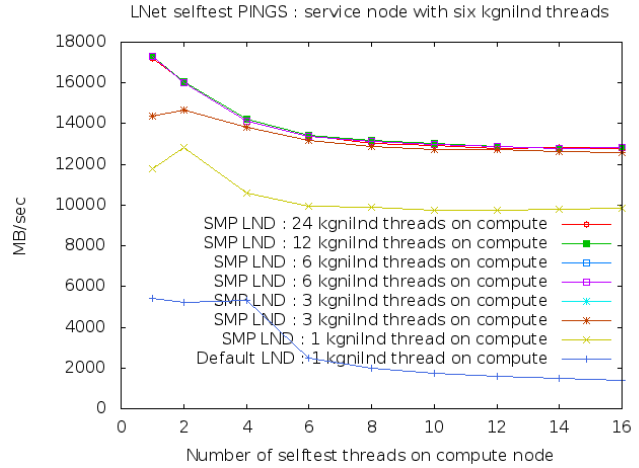

The data for the 4K sized messages shows a much more rapid convergence of the network transfer rates





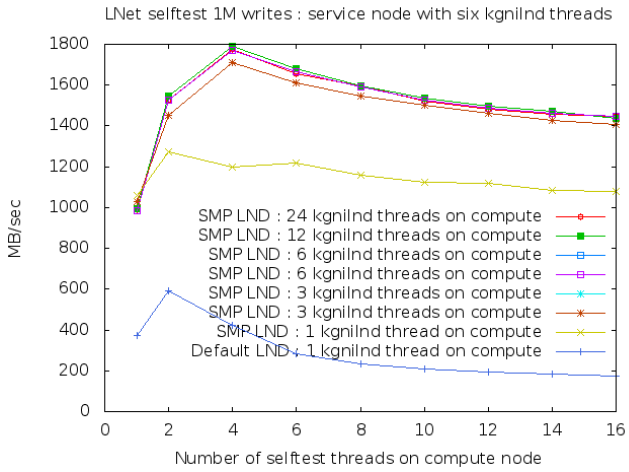

With smaller messages the cost is less so convergence happens at a more rapid pace then with bulk transfers. As is the case with the default driver the lack of support for CPU affinity for the threads negatively impact any improvements we might expect by adding more threads to any system. As we seen in the compute to compute case a heavy cost occurs with a larger number of threads being migrated between cores. Even in the default driver when running six threads on the service node we see the three thread setting on the compute nodes do slightly better than the others.

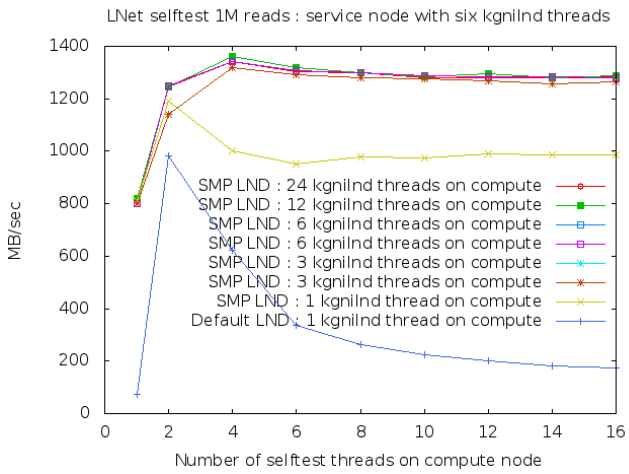
This data repeats the previous behavior we seen in the compute to compute case. If the trend is to be followed then we should see the opposite behavior on the SMP enhanced system. As the number of threads are added no penalties will occur but we will experience diminishing return with each increase in the number of the threads in the system. Also expected is that the performance of the SMP driver will be greater than the default LND driver. The question that needs to be answered does the three thread case on the compute node give a edge over using a single thread . If that is true then we know over all it is safe to increase the thread count on the compute node to three. The first hint to this behavior can be observed with the LNET ping test. The data reveals that once again that the SMP version in this case doubled the performance. Going from one to three threads on the compute gave another thirty five percent increase.




Increasing the thread count any further gives a increase only for a small number of LNET self test threads. With enough self test threads we see the network performance converge with all the other high thread count cases.

While we see a convergence to a lower value for LNET pings data but for read and write performance this is not the case. The values recorded at the lowest concurrency don't start out at the highest value as is the case for LNET pings but the lowest. A maximum is reached between four and six LNET self test threads. This is expected since we are running with six threads on the service node. Depending on the data we are seeing several factors of increased performance. The data supports that using three threads on the compute node helps to push more data to the router.

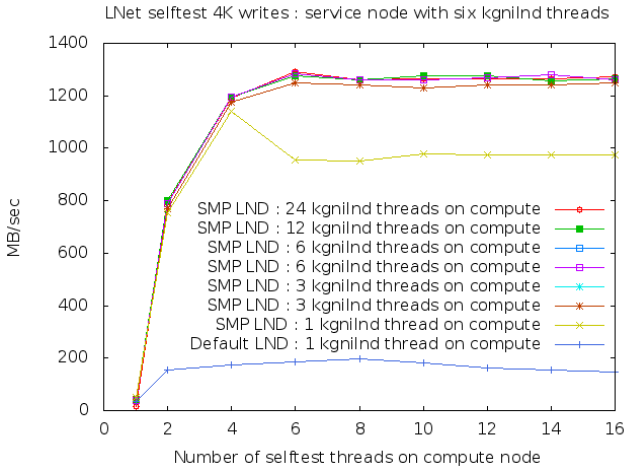


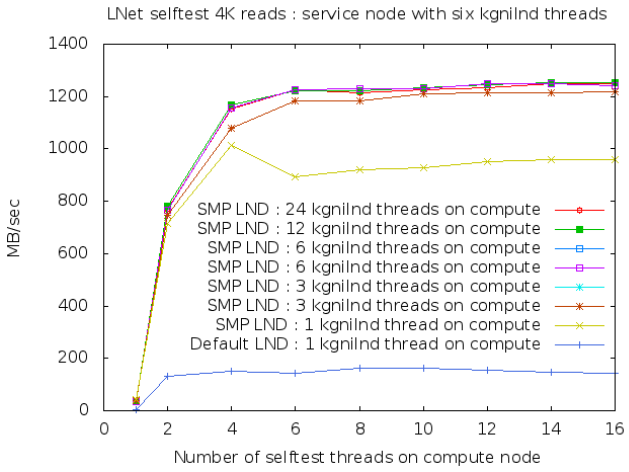




Using the local cache of the core the thread is running on removes the contention that we see in the default case. Some performance loss in the single thread case for SMP still occurs. In this case the routers own threads are under utilized which is causing the dip we see.

Small size packets track very closely the bulk message behavior. In the default driver we see no contention

but also missing is any performance. The amount of data being received just flat lines.

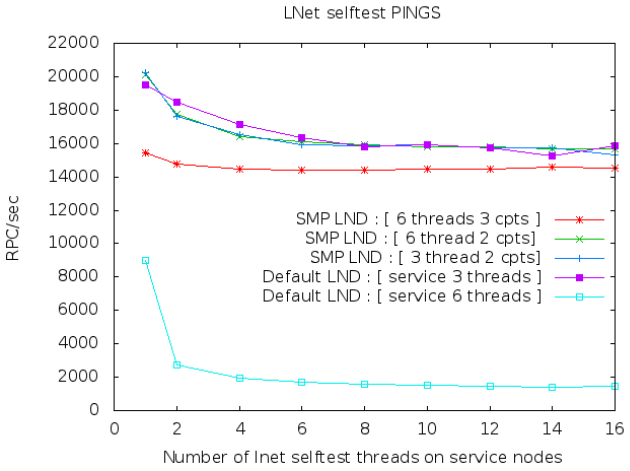


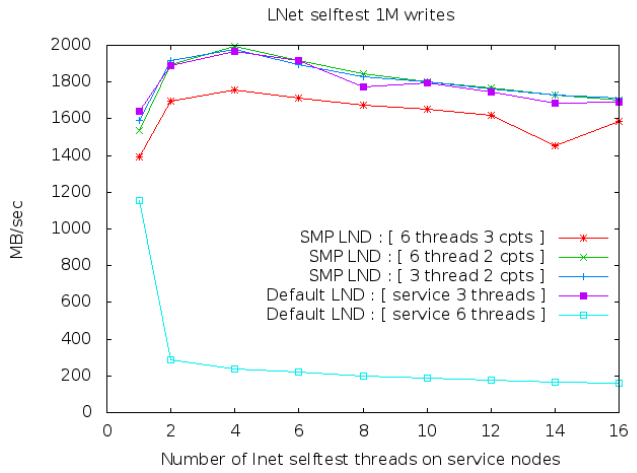

*3) Service node to Service node*

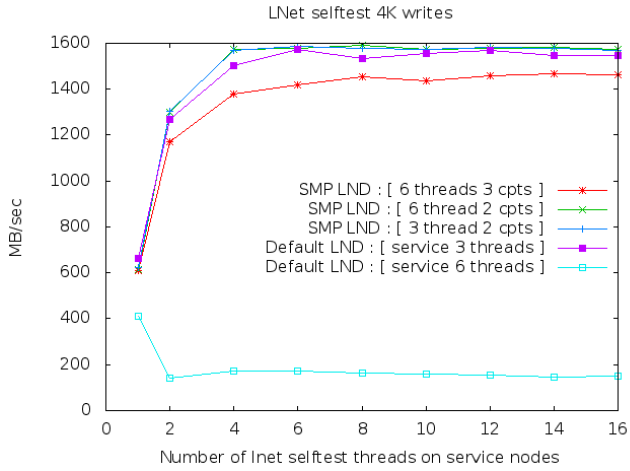
Our last test scenario deals with service node to service node communications. While not the most critical path in the system we do encounter this configuration. It also gives us more insight into how service nodes are affected by the SMP scaling changes. Due to the service node hardware we have less limited configurations to test than the compute nodes. In this case we can directly compare the SMP and non SMP versions of the LND driver in the same figure.

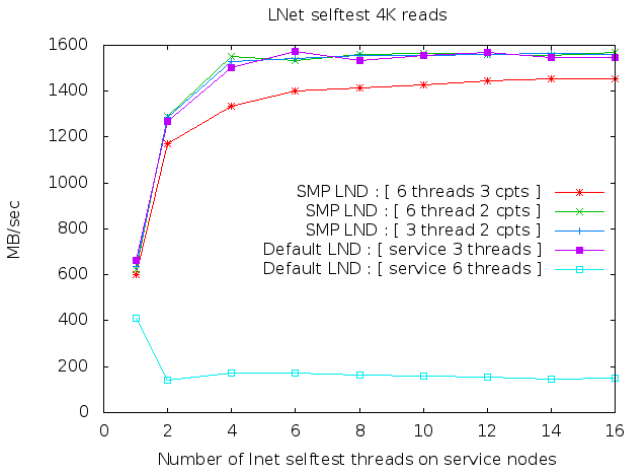
As in the previous service node analysis we have the options of three or six threads. Included are cases of using different size CPTs. In previous test we seen adding another CPT decreases the performance.




Much like the compute node to service node results we see the behavior of the LNET pings experiencing a small decrease at higher concurrency. In all cases the performance is very consistent.






This concludes our examination of our Gemini system. Even with these results being specific to our hardware it helps lay down the how to best take advantage of your equipment. Even on different hardware we expect that only a small number of threads would be needed on the compute nodes.

## Future work

A great amount of data was collected for this project but there are more options to be explored. Currently the code in the patch is written to spawn a new thread in the same CPT until it is filled. To help parallelism in the LNET stack better each new thread could be spawned on a different CPT. We briefly touch on the checksumming issues. The truth is they are some of the biggest obstacles to reach our performance goals. In future work different check summing methodologies algorithms will be tested to see if we can improve performance. Lustre versions 2.3 and above provides an infrastructure to take advantage of the hardware to accelerate check summing. With these new algorithms Lustre also applies them to the ptlrpc layer. Currently the Gemini LND driver does check summing of bulk messages as well as the ptlrpc. Double check summing is a waste of valuable cycles. Eliminating this double checking will help cut the loss in performance to services provided to LNET users.