Scalability Analysis of Job Scheduling Using Virtual Nodes

Norman Bobroff[1], Richard Coppinger[2], Liana Fong[1], Seetharami Seelam[1], and Jing Xu[3]

[1] IBM T.J. Watson Research Center, Hawthorne, NY 10532, USA
{bobroff,llfong,sseelam}@us.ibm.com
[2] IBM Systems and Technology Group, Poughkeepsie, NY 12601, USA
coppinge@us.ibm.com
[3] University of Florida, Gainesville, FL 32611, USA
jxu@acis.ufl.edu

Abstract. It is important to identify scalability constraints in existing job scheduling software as they are applied to next generation parallel systems. In this paper, we analyze the scalability of job scheduling and job dispatching functions in the IBM LoadLeveler job scheduler. To enable this scalability study, we propose and implement a new virtualization method to deploy different size LoadLeveler clusters with minimal number of physical machines. Our scalability studies with the virtualization show that the LoadLeveler resource manager can comfortably handle over 12,000 compute nodes, the largest scale we have tested so far. However, our study shows that the static resource matching in the scheduling cycle and job object processing during the hierarchical job launching are two impediments for the scalability of LoadLeveler.

1 Introduction

Job scheduling software is a key piece of system software to maximize the utilization of parallel computing systems. As these systems increase in size with one generation of systems having more processors and compute power than the previous generation, the performance of job scheduling becomes crucial to optimize the overall system utilization. To support the current and next generation of massively parallel systems (MPP), the job scheduler must scale in several dimensions. It must be able to manage a large number of jobs and compute resources, quickly match resources to a job, and rapidly dispatch the job on those resources. During the last year, we have analyzed the scalability of IBM Tivoli workload scheduler LoadLeveler in the context of the IBM DARPA HPCS program [2]. This paper presents the method and results of the study, as well as insights about scheduling and dispatching in the context of LoadLeveler.

An essential requirement for a scalability study is access to a representative large scale system. Although we indeed have access to fifty 8-processor pSeries machines, the goal is to study scalability beyond a thousand nodes. This resource limitation is overcome by developing a lightweight virtualization mechanism that creates hundreds of LoadLeveler nodes on each physical machine. This technique allows testing on up to a 12,000 node LoadLeveler cluster using fifty physical nodes. Virtualization is applied to study the scalability of LoadLeveler in its capability in resource monitoring, identifying and scheduling jobs to resources, and in dispatching jobs to the allocated resources. The following contributions are made:

- Introduce lightweight virtualization technology that isolates and executes multiple instances of LoadLeveler node daemons on a physical machine, creating a large cluster with minimal physical machine and memory requirements (Section 3).
- Analyze the scalability of job scheduling algorithms for sequential and parallel jobs and isolate the performance of different phases in the scheduling algorithm. Static resource matching is determined to be a scalability problem and approaches to address this problem are described (Section 4.2).
- Investigate the scalability of hierarchical job launching and identify scalability hot-spots with processing job object at various levels of the hierarchical tree (Section 4.3).

2 LoadLeveler Overview

LoadLeveler [3] is a distributed job scheduling product of IBM. It is based on the licensed code from the Condor system [13] in mid 1990. The architectural framework of LoadLeveler, as shown in Figure 1, retains the core structure of Condor. The *Central Manager* (CM) consists of two functional units: the *Collector* and the *Negotiator*. The Collector receives resource information sent by a daemon called *StartD* running on the machine.[1] LoadLeveler ensures there is only one StartD on each machine with responsibility for reporting the machine state, resources and attributes, utilization, and managing presence heartbeats. The Negotiator applies this resource information to allocate machines matching the execution requirements of user jobs.

Jobs are submitted to LoadLeveler through the *Scheduling daemon* (SchedD). SchedD is responsible for maintaining a local job queue and persisting job state, as well as coordinating the activities of assigning execution nodes to the job, and launching the job. Multiple SchedD machines can be defined for a cluster to eliminate bottlenecks with large job submissions requirements. SchedD informs the Negotiator about each job arrival and the Negotiator applies scheduling algorithms to allocate computational and other resources to jobs. The resource assignments are returned to the SchedD which forwards the job launch information to the StartDs on the allocated executing machines. Execution at the node is managed by the local StartD which forks a *Starter* process to initiate and control the job. Concurrent execution of multiple jobs or tasks at the node is enabled by forking multiple Starters.

[1] The terms ”machine” and ”compute node” are used interchangeably.

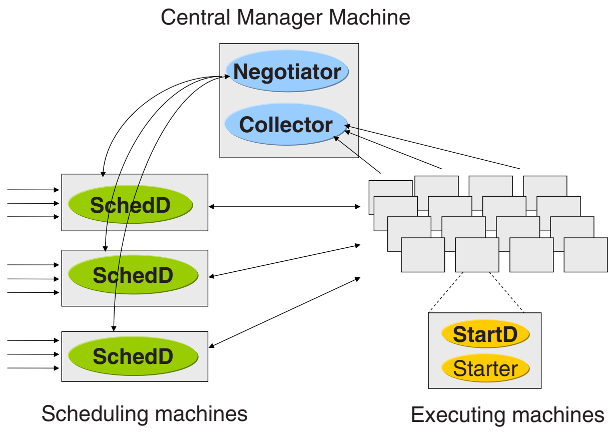

Fig. 1. LoadLeveler Architecture

LoadLeveler continually evolves to support new hardware architectures, and leverage novel software features. Examples include high bandwidth interconnection switches (e.g. SP2 switch [12], InfiniBand [10]) hardware multi-threading, and Blue Gene [6,5]. Supported software features include AIX's WLM [1], and various Linux distributions [4]. Recent development has emphasized support for highly parallel applications running on large scaled clusters with high speed interconnection instead of flocks of workstations [7].

The following sub-sections describe in more detail scheduling in Negotiator, and hierarchical communication scheme for scalable job dispatching. This material provides the necessary background for our scalability studies covered later in this paper:

2.1 Scheduling in Negotiator

The Negotiator of CM processes incoming jobs in two sequential phases: scheduling requested resources to jobs and coordinating with SchedD to dispatch jobs to assigned machines for execution.

During the scheduling phase, Negotiator logically performs the following steps:

1. Select machines that have the capabilities to match the requirements of the job; Exemplary capabilities include machine architecture type (x86, POWER), job class definitions, and high-speed switch connectivity.
2. Select capable machines that have the dynamic capacities to assign to the job; Exemplary capacities include unused job class, unfilled multiprogramming level, and spare switch adapters.
3. Assign the machine(s) to the job based on specific scheduling algorithms and administrative policies; for example, backfill [9] and fairshare are two of scheduling algorithms in LoadLeveler.

At the end of scheduling phase, Negotiator sends successful machines-to-job assignments to SchedD, which dispatches the job to all assigned machines through the StartDs on the machines.

2.2 Hierarchical Scheme for Job Dispatching

In a large cluster, information and command propagation from a single machine to a large number of other machines is parallelized (e.g. using multi-threading) for faster communication. However, the number of open communication connections on a single machine is a potential bottleneck because of limited resources such as communication buffers and OS data structures. The hierarchical scheme provides the benefit of parallel communication operations by dividing the connections through a spanning tree.

The hierarchical communication scheme implemented by LoadLeveler [8] is shown in Figure 2 with an exemplary fan-out of three. When a job is scheduled by the Negotiator a job object of assigned machines and job specific information is sent to SchedD. SchedD constructs a hierarchical spanning tree of machines using the configurable fan-out parameter. and sends the tree structure and job information to the root node or master StartD. The master StartD repackages the job object into job objects customized exclusively for the subtree headed by each immediate child StartD and forwards the information. This process is repeated down the tree to the leaf nodes. Communication of the job object to all compute nodes is flagged as successful when an acknowledgement from each node in the tree is received at the master StartD. Then, a subsequent communication is sent using the tree to command each node to start the job. The start command prompts each StartD to launch a Starter process which locally manages job execution.

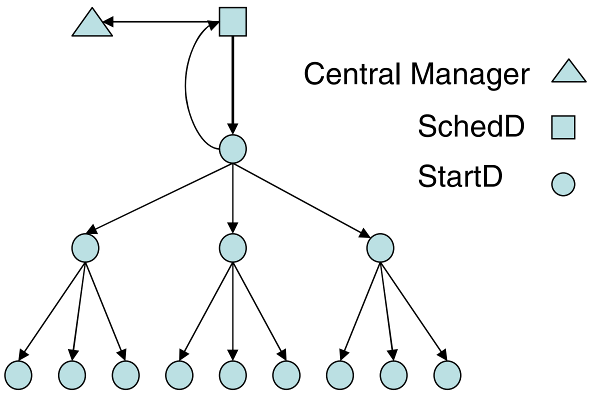

Fig. 2. Hierarchical Communication Structure

3 Virtual Node Method for Scalability Analysis

Analyzing the performance and scalability of LoadLeveler requires a cluster of at least a few thousand compute nodes. Building and maintaining such a large scale cluster for this analysis alone is not cost effective. A StartD instance is the component in LoadLeveler that represents a compute node. From the perspective of the CM the size of the cluster is the number of StartD processes reporting to the CM. Thus, the key to creating a compute cluster whose apparent size is greater than the number of physical machines is to allow multiple StartD to execute independently and in mutual isolation on each physical node. However, the design point of LoadLeveler is a single StartD process for each physical machine node. Communication ports are the identical for all StartD as specified in a central configuration file and can not be shared among StartD instances on the same machine. Furthermore, each StartD must appear to both the CM and other StartDs that it is located at a dedicated and unique ip address. A secondary issue is that the configuration file read by each StartD also contains information about spool and log file locations which cannot be shared between StartD. So the challenge is to provide an environment to StartD where it is on a private network.

One approach is to fully virtualize the platform hardware at each physical compute node using a hyper-visor to execute multiple operating system images, each running a single StartD. This provides the requisite isolation of network bindings and configuration settings. The drawback is the memory, disk, and processor overhead of using the operating system as a isolation container. A StartD process requires about 25MB of memory including the Starter process, so an OS container is inefficient. A better approach is lightweight virtualization of just the network layer so that multiple StartDs execute in isolation within a single OS image. It is reasonable to expect that a lightweight solution on a physical machine with 4 GB of memory is capable of hosting approximately 160 'virtual' compute nodes, more when using memory swap space on disk. A further practical challenge to lightweight virtualization is that the initial implementation cannot involve modification of product code. The product group is willing to make minor modifications to network interface binding to simplify network isolation, when substantial benefit of emulating large clusters is demonstrated. Such benefit is initially displayed first using an IPTables approach which motivated a minor (¡ 10 lines of code) change to the IP binding in LoadLeveler as explained next.

The adopted methodology for lightweight virtualization of StartD nodes is now described. The initial hurdle is to generate a unique configuration file for each virtual StartD. Activation of a StartD process causes it to read a configuration file from a fixed location. The configuration file defines daemon communication intervals, locations for log files, spool directories, and IP ports on which StartD listens for external traffic. Subsequently StartD spawns a thread that binds to its set of listening ports. As part of job launch, StartD also forks off Starter processes which also rely on this configuration file. Thus, the configuration file cannot be modified while the StartD process is active. Fortunately, the LoadLeveler development team identified an an environment variable

LOADL_CONFIG which when set is recognized by StartD as an override to the global configuration file. The original intent of this variable is to manage jobs in multi-cluster LoadLeveler environment [3]. It is used here to provide each StartD with a unique set configuration parameters by setting LOADL_CONFIG to a corresponding file prior to instantiating the StartD process. However, the network conflict issue remains as the StartDs still share a common IP address and conflict over the binding to communications ports.

Tying the communication of each StartD to a different IP requires a flexible method to create multiple IP addresses and use them for CM, SchedD, and StartD communications. Because code modification is not permitted in the proof of concept phase, the approach taken to network isolation is based on iptables. Iptables maps ports and addresses at the ip layer of the transport stack. For a large scale system this requires complex setup and imposes significant performance overhead. The iptables approach is interesting and a potential solution to other lightweight virtualization problems [11]. It is successfully used to demonstrate value of virtualized StartD to the product group, but because it is not used as an ongoing solution the discussion is presented in the appendix A.

The adopted solution, implemented with minor code changes, introduces a private network between the CM node, SchedD node, and the compute nodes. Many network performance related studies use this method of creating a private network. The basic idea is to create IP aliases to the network interfaces of the physical machines hosting CM, SchedD and StartDs. The alias adapters appear in the output of the Linux command /sbin/ifconfig. All aliases are created within the same subnet so that the machines can communicate without routing. A single alias of the network interface is built for the CM and another for the SchedD node. On StartD nodes, an alias of the network interface is created for each StartD that runs on the machine. Since LoadLeveler uses hostnames to map to ip addresses, a unique hostname is assigned to every IP address and the hostname to ip mappings are placed in the /etc/hosts file of every machine. For example, to execute 500 StartDs on a physical node 500 network adapter aliases with consecutive ip addresses are created. An alias for the CM and SchedD leads to a total of 502 entries in the /etc/hosts files of the physical nodes. Having large /etc/hosts file has performance implications discussed in the experimental section 4.

With the aliased network adapters in place, communications endpoints need to be bound from every StartD to a corresponding aliased ip address. The original code binds an endpoint to the global listening port specified in the configuration file and physical machine ip address. The modified code issues a bind to the global listening port and an ip address from an aliased network adapter. The aliased adapter hostname for each StartD is passed from a new environment variable (LOADL_HOSTNAME). With this modification all StartD instances use the globally specified listening ports, together with unique ip addresses. Figure 3 summarizes the communications setup. Less than 10 lines of code are needed to support this expanded endpoint binding functionality. The procedure isolates

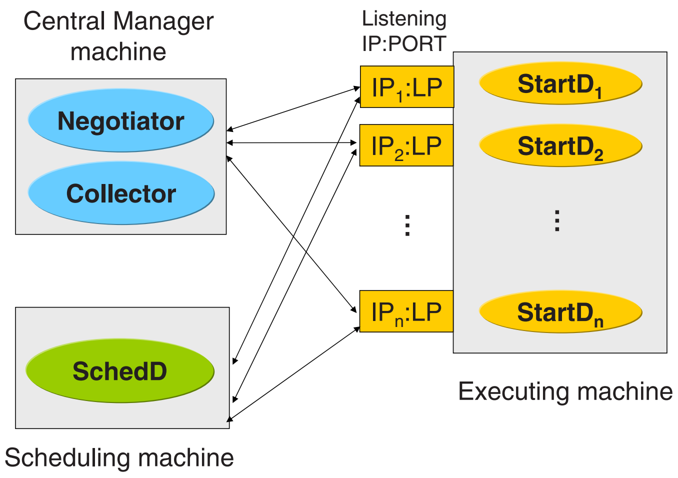

Fig. 3. LoadLeveler with virtualized StartDs – Binding to IP:Port combination

StartDs from each other while preserving common listening ports, and makes them appear to the central manager as distinct machines.

Virtualized StartDs provide a lightweight platform for studying scalability but have other limitations. Each StartD thinks it manages all resources of a node thus, the total amount of resource reported at the CM is the actual resource on the node multiplied by the total number of StartDs. As a result, the quantity reported at CM may not be used to study the turn around time of real jobs that require some amounts of static and dynamic resources on the nodes. This study is restricted to the scalability of scheduling at the Central Manager and hierarchical job launching.

4 Performance and Scalability Studies

The performance of LoadLeveler scheduling and job dispatching is evaluated in two interesting limiting cases: 1) A parallel job requiring all compute nodes in the system. 2) A single node job on an occupied cluster. The study explores the applicability of the multiple StartD per node approach and more importantly identifies bottlenecks in the LoadLeveler implementation that limit scalability. The study conveniently separates into two sections along the lines of LoadLeveler functions. Job processing from submission to launch at the compute nodes consists of the sequential and independent cycles of scheduling and dispatching. According to LoadLeveler implementation (Section 2), the former occurs in the Negotiator which executes on the CM machine, while the latter is distributed involving communicating processes on SchedD and between the StartD on each compute node. Based on our analysis, possible solutions are suggested to mitigate scalability problems and improve the performance issues.

4.1 Methodology

LoadLeveler is treated as a black box and performance data is extracted from log file messages which are recorded with microsecond timing. Care is taken that logging does not interfere with system performance as certain flags cause log performance to overwhelm actual function. Because the granularity of control of debug flags is coarse an option such as `D_NEGOTIATE` generates thousands of messages for a single scheduling operation in a large system. Although each log event takes about two microseconds to time-stamp and format and is written in the background the aggregate effect causes noticeable delay. The log file data is used to verify that log events do not impede scheduling operation.

The job description used to drive the experiments contains matching constraints on job class and the number of nodes. Typically, the executable is a shell script that invokes a 60 second `sleep` command. An MPI executable invoking `sleep` on each node is also used.

Experiments are performed in two different cluster computing environments. One is a homogeneous collection of 50 IBM pSeries 575 machines running IBM AIX version 6 and connected by a high speed SP switch. Each machine has eight dual core IBM POWER5+ processors and 32GB of memory. A heterogeneous and smaller cluster of machines is also used to collect data. This cluster typically contains 6 to 8 compute nodes consisting of partitions on an IBM pSeries 575 and IBM POWER blade servers connected by a ten gigabit ethernet switch. The blades have two dual core IBM POWER5 processors, 4 to 16GB memory, and run Red Hat Enterprise Linux version 5. Each cluster dedicates a machine for CM (scheduling and resource manager) and SchedD (dispatching and job life cycle management). Each StartD is configured to have a single Starter so that only one task is executed per virtual node.

4.2 Scheduling Analysis

The time to schedule a job depends on the number of nodes, the parallelism of the job, and the complexity of matching job requirements to compute node resources. The study starts by quantifying the dependence of single node job scheduling time on the number of compute nodes, then moves to large parallel jobs. In both cases the compute nodes are unoccupied and a single job is placed in the LoadLeveler queue. A timing event is issued in the log file when the assignment of the job to a compute node is complete and the job is dequeued from the submit queue.

Single node job scheduling analysis. Figure 4 shows scheduling time for a single node job where the independent variable is the number of compute nodes (N). Because the nodes are initially unoccupied, the expected result is that scheduling time is independent of the number of N, instead, the data show a linear dependence on N. This result is understood in terms of the three scheduling steps described in Section 2. First, the scheduler scans all machines to locate candidate nodes that have the capability to execute the job. The capability scan is an O(N) operation that produces a bit map of all machines. The bit map is applied in the subsequent machine matching steps.

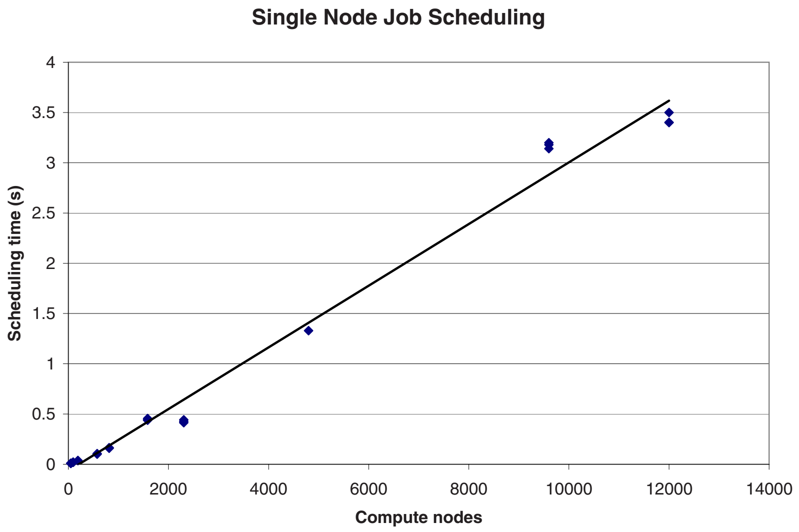

Fig. 4. Scheduling time for a single node job

The second and third scheduling steps assign the job to the compute node with the highest priority. In LoadLeveler, the priority of each compute node is defined by the system administrator using a formula syntax. The default priority scheme, used here, decreases the priority of a node linearly with the short term (five minute) average processor utilization of each machine. The scheduler maintains a priority sorted list of all compute nodes. List order is updated when jobs are scheduled or terminated as well as by periodic updates of node resource consumption reported by StartDs. The list entry for each compute node is a summary of the latest reported resource consumption information about the nodes, e.g. utilization, memory, job class, network adapters, multi-programming level. In the second step, the scheduler takes the top priority node and checks the corresponding bit map entry from the first step. If true, the latest resource information is checked to see if the node has the current capacity to execute the job. If not, the scan of the compute node list continues until a match is found. A machine match results in a job assignment to that node. In this experiment, the second step time is constant time because the cluster is unoccupied and the machine at the top of the priority list is always available.

Thus, the single node scheduling time is determined by the O(N) behavior of the first step of the scheduling process. The data in Figure 4 confirm this and show how virtual StartDs enable experiments on the 50 physical node cluster to 12,000 nodes. The data are linear and the scheduling time per node is about 250us for the static resource matching step. Furthermore, the resource manager component of the CM performed remarkably well as 12,000 nodes are brought up, identifying all resources in less than two minutes.

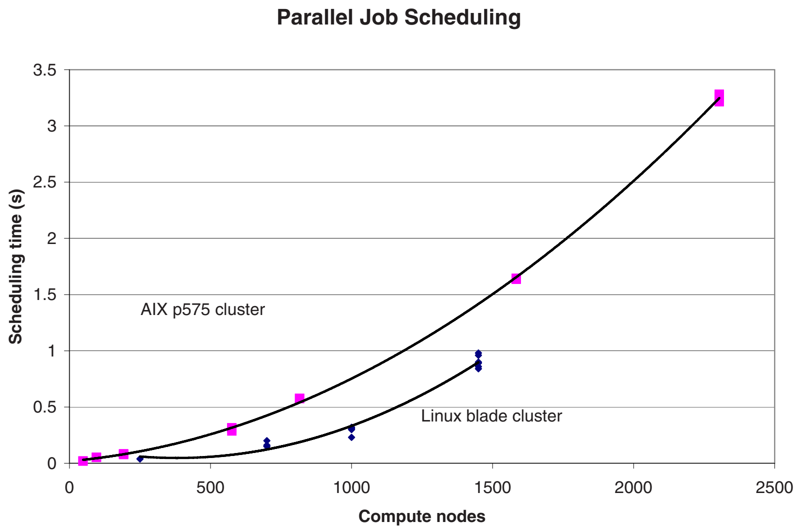

Fig. 5. Scheduling time for parallel job requesting all compute nodes

This result points to the importance of improving the performance of the capability scan step of the scheduling cycle. However, it is useful to recognize that in many scheduling conditions, generation of the bit map in the capability scan is an optimization. The resource matching of the second step is more extensive and costly than that of step one and should be avoided for nodes that cannot potentially execute the job. Such a situation occurs with heterogeneous compute resources as in an ad hoc cluster of workstations where the bit map may be sparse, significantly reducing the number of machines tested in the dynamic matching of the second step. Also, in a highly utilized cluster there is a good chance that a job is not initially scheduled and needs to be retried. The bit map is retained with the job and is not recreated in subsequent cycles.

There is still room for enhancements. One observation is that the capability scan step can be decoupled from the scheduling cycle. Results of the scan are based on static information about the jobs and compute nodes. So it can be performed when a job arrives at the CM prior to the scheduling cycle. Binding the capability scan to the scheduling cycle makes sense when the scheduler supports ad hoc clusters of workstations with intermittent availability or connectivity. A decoupled bit map can become out of sync with the cluster state. But this is not an issue for high performance compute clusters.

A further observation is that clusters are frequently homogeneous. In this case, every bit in the capability map has the same result for a given job. Here, the scan result has no added benefit to the machine list matching step. So the scheduling cycle can have distinct operational states depending on the heterogeneity. Selection of the state is inferred dynamically based on cluster workload. When recent job submissions generate a bit map that is largely ones, the scheduler folds the capability scan logic of step one into the matching of step two which eliminates the need to scan the entire cluster. If the machine assignment step fails frequently, the capability scan step is reconstituted as a distinct step to regain the advantage of having a single scan for multiple scheduling cycles.

Parallel job scheduling analysis. The next study investigates a parallel job requesting all N cluster nodes. This study targets the second scheduling step because the capability scan is performed once while machine matching and job assignment are exercised N times.

In this experiment, the system is initially unoccupied and a single job requesting all nodes is submitted. Figure 5 shows results for the two clusters. The first observation is that the data for the both clusters has a second order component that significantly impacts performance above 1000 nodes. This behavior is unexpected as the scheduling steps should be linear in the number of nodes.

4.3 Dispatching

As described in Section 2.2, SchedD starts the job dispatch process by constructing a *job object*. All information necessary to execute the task on the assigned compute nodes is contained in the job object. While many task details such as the binary executable location are common to all nodes, node specific details such as which network interface to use are also included. The job object also contains the structure of the hierarchical communication tree used to distribute and communicate job information and status between the compute nodes and the SchedD. The job object is forwarded from SchedD to the master StartD to initiate job dispatching. Subsequent responsibility for constructing the communications tree and propagating job dispatching information from the master StartD node to the compute nodes on the tree belongs to the StartDs at each compute node. The StartD has no *a priori* information about the tree structure. It decodes the job object passed to it and locates its children. Then, a new job object is created for each child customized to contain only information required for the child's subtree. This process continues until the tree is fully constructed.

Dispatching performance is studied in a large scale environment created by running multiple instances of StartDs on each compute node. The starting point is to establish the equivalence of logical and physical StartDs within the context of hierarchical communication. There is extensive communication between StartD instances during the job dispatching cycle. The pattern and concurrency of the communication processing is expected to change when multiple StartD execute on a common physical platform. For example, the concurrency is limited by the number of available free processing units. Validation starts by comparing a fully parallel cluster with a single StartD on each of 48 nodes to a single physical node with 48 instances of StartD.

The first experiment compares dispatch time as a function of the fan-out. The measured dispatch time is the interval that starts when the master StartD receives the job object and ends with an acknowledgement at the StartD from all nodes on the tree. The results are presented in Figure 6. The lower solid line corresponds to

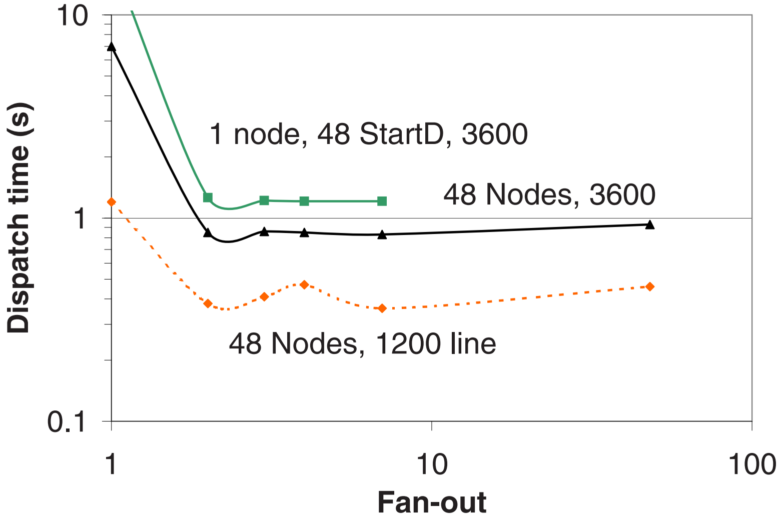

Fig. 6. Compare logical to physical StartD at 48 nodes. The 1200, 3600 indicate size of /etc/hosts file.

the fully parallel cluster of 48 machines and the '1200' on the label indicates the number of entries in the /etc/hosts file. The intermediate dashed line is the same experiment as the lower line except that the /etc/hosts file has 3600 lines. Data for the single machine, multiple StartD case is at the top.

The qualitative behavior of dispatch time is expected based on the tree architecture. The number of levels L in a tree of fan-out F with N nodes is

$$L = \left\lceil \frac{\log(1 + N(F-1))}{\log(F)} \right\rceil .$$

Performance is poor for a fan-out of unity as this degenerate case serializes communication over the 48 levels. A binary tree reduces the number of levels from 48 to 6 and the multi-processor platform concurrently executes StartDs. Additional benefit is expected as fan-out grows because of concurrency and logarithmic reduction in tree depth. The trend is expected to reverse when fan-out exceeds machine concurrency and communication becomes serialized.

Figure 6 demonstrates that for up to 48 StartD the behavior of the two configurations with fan-out is comparable. This is an important step in validating the virtual StartD methodology that allows many StartD on each physical node. However, the expected gain from increasing fan-out beyond a binary tree is absent. This suggest some operation is limiting performance and is largely independent of the fan-out or depth of the communication tree. The log file is investigated for more information, but the 48 node job does not provide a clue to the cause.

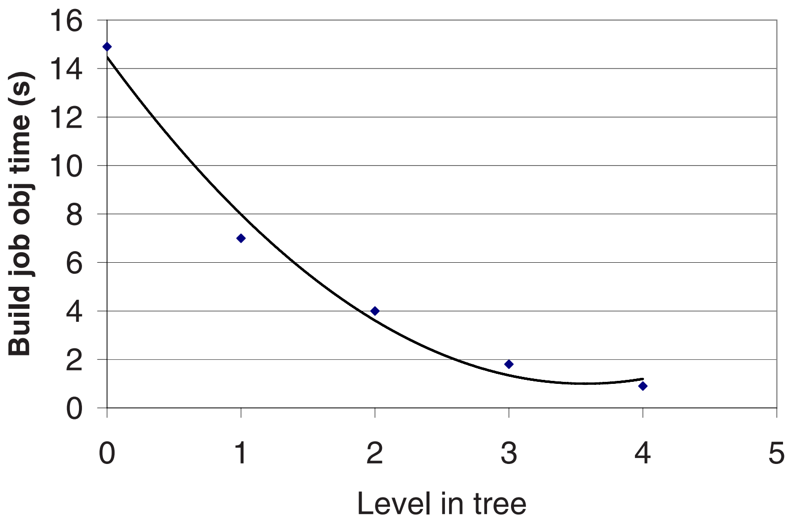

Fig. 7. Time to build job object, 2304 node binary tree

In an effort to identify the origin of the problem, a parallel job requesting 1000 nodes is executed with fan-out of 2 in the environment of 2304 compute nodes by running 48 StartD on each of the 48 physical nodes. The StartD log file shows significant processing occurs at each StartD prior to forwarding the modified job objects to the F children. In particular, the processing time is proportional to the number of compute nodes remaining on this branch of the tree. This decreases approximately as 1/F at each level down the tree (i.e. as N, N/F, ..., N/F^L, for the L^{th} level), as shown in Figure 7. It is apparent that time spent de-serializing the objects from the communications buffers and repackaging and serializing job object for each child overwhelms the potential performance gains expected from increased fan-out. To support heterogeneous clusters the external data representation (XDR, RFC-1832 (1995)) standard is used to encode/decode every field in the job object and is a large component of the observed overhead. Because most fields are read for the sole purpose of repackaging and copying from parent to child there is no reason for these fields to be decoded/encoded at each level of the tree. Structuring the HC messages to substitute much of the XDR activity with buffer copies is a potential optimization.

This discovery shows that attention to all aspects of a hierarchical communications scheme is required to achieve the anticipated gains. The layout of the data structure used to transfer information in the tree needs to be easy to parse and rebuild. This raises the question of whether a job object format is possible so that decoding and repackaging occurs in F rather than N operations. The cost of the initial construction remains proportional to N, but is incurred once at the SchedD machine instead of on the compute nodes.

This example of performance problem detection and analysis highlights the advantage of using lightweight virtualization to create a large scale system for testing job scheduling and launching. It exposes design issues not apparent at the physical cluster size available to developers. The 2304 node cluster is leveraged further by measuring the dispatch time for a 2304 node job with the tree structures of fan-out 2 to 13 with levels of 12 and 4, respectively. The time is about 100 seconds compared with 60 seconds obtained by linear extrapolation from the 48 node system of Figure 6. This is not an unreasonable prediction error for a factor of 48 scale up in a computer system.

An additional performance consideration revealed in the experiments related to Figure 6 is the effect the /etc/hosts file size. The IP alias for multiple StartDs on a node require entries in the hosts file increasing the processing time of IP lookup. Investigation using the secondary test cluster suggests that significant time is spent searching the /etc/hosts file for IP resolution. The figure demonstrates the 48 physical node data is significantly improved when /etc/hosts is reduced to 1200 from 3600 lines. Unfortunately, for large systems the size of /etc/hosts is considerable large. The speed of /etc/hosts lookups is also operating system dependent.

5 Concluding Remarks

A lightweight virtualization methodology is introduced to LoadLeveler and applied to study the scalability of job scheduling and dispatching in large scale parallel systems using modest number of physical nodes. The study identifies static resource matching in scheduling and job object processing in dispatching as potential scalability bottlenecks. and proposed solutions to their performance. Further research needs to be applied to investigate whether results observed here continue to demonstrate the same functional scaling in larger systems.

Acknowledgments

We are extremely grateful to Linda Cham for tirelessly helping us with our most last minutes requests to build the LL library for various platforms. We also thank Waiman Chan, Alexander Druyan, Nathan Falk, Kevin Gildea, Kailash Marthi, and Ravindra Sure for numerous valuable discussions. We express our gratitude to David Olshefski and Gong Su for their valuable insight on iptables setup. This material is based upon work supported by the US Defense Advanced Research Projects Agency (DARPA) under its Agreement No. HR0011-07-9-0002.

References

1. AIX 5l workload manager (wlm), `http://www.redbooks.ibm.com/redbooks/pdfs/sg245977.pdf`
2. Darpa high productivity computing systems project, `http://www.darpa.mil/ipto/programs/hpcs/hpcs.asp`

3. IBM tivoli workload scheduler loadleveler, `http://publib.boulder.ibm.com/-infocenter/clresctr/vxrx/index.jsp`
4. Linux distributions, `http://www.linux.org/dist/`
5. Aridor, Y., Domany, T., Goldshmidt, O., Kliteynik, Y., Moreira, J., Shmueli, E.: Open job management architecture for the Blue Gene/L supercomputer. In: Feitelson, D.G., Frachtenberg, E., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2005. LNCS, vol. 3834, pp. 91–107. Springer, Heidelberg (2005)
6. Aridor, Y., Domany, T., Goldshmidt, O., Kliteynik, Y., Shmueli, E., Moreira, J.E.: Multitoroidal interconnects for tightly coupled supercomputers. IEEE Trans. Parallel Distrib. Syst. 19(1), 52–65 (2008)
7. Pruyne, J., Livny, M.: A worldwide flock of condors: Load sharing among workstation clusters. Journal on Future Generations of Computer Systems (1996)
8. Moreira, J.E., Chan, W., Fong, L.L., Franke, H., Jette, M.A.: An infrastructure for efficient parallel job execution in terascale computing environments. In: Supercomputing 1998: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM), Washington, DC, USA, pp. 1–14. IEEE Computer Society, Los Alamitos (1998)
9. Mu'alem, A.W., Feitelson, D.G.: Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. IEEE Trans. Parallel Distrib. Syst. 12(6), 529–543 (2001)
10. Pfister, G.F.: An introduction to the InfiniBand architecture. In: Jin, H., Cortes, T., Buyya, R. (eds.) High Performance Mass Storage and Parallel I/O: Technologies and Applications, ch. 42, pp. 617–632. IEEE Computer Society Press/Wiley, New York (2001)
11. Ryu, K.D., Daly, D., Seminara, M., Song, S., Crumley, P.G.: Agent multiplication: An economical large-scale testing environment for system management solutions. In: IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, April 2008, pp. 1–8 (2008)
12. Stunkel, C.B., Shea, D.G., Aball, B., Atkins, M.G., Bender, C.A., Grice, D.G., Hochschild, P., Joseph, D.J., Nathanson, B.J., Swetz, R.A., Stucke, R.F., Tsao, M., Varker, P.R.: The sp2 high-performance switch. IBM System Journal 34(2), 185–204 (1995)
13. Tannenbaum, T., Wright, D., Miller, K., Livny, M.: Condor – a distributed job scheduler. In: Sterling, T. (ed.) Beowulf Cluster Computing with Linux. MIT Press, Cambridge (2001)

A Appendix Iptables for StartD Virtualization

Iptables is an approach to orchestrate the communication between the CM, SchedD and StartDs without code modification. It is a generic network packet manipulation technology that enables packet filtering, network address translation, and packet mangling. Iptables is used extensively in building Internet firewalls, redirecting traffic between servers, sharing public IP addresses.

The difficulty with StartD is that its IP endpoint sockets are bound to the listening port specified in the 'LoadL-config' file and the ip associated with the machine hostname in the network interface. Thus, while each StartD within a physical machine is assigned a unique listening port by directing it to a unique configuration file (e.g. 'LoadL-config.nnn') the ip address of the endpoint listen socket binding is hard coded. This is fine as the StartD listeners within each physical machine no longer conflict with each other. However, the CM and SchedD assume all StartD in the cluster listen on the same port number at each machine. What is required is a way for the multiple StartD on each physical machine having unique ports but a shared ip to appear to the CM and SchedD that they are at common port but unique ip addresses. In concept, the resolution is to use iptables to:

– Map outgoing packets from the StartD endpoints (unique-port, common-ip) to appear to originate at (common-port, unique-ip).
– Map outgoing packets from CM and SchedD sent to (common-port, unique-ip) to be sent to (unique port, common-ip)

This is accomplished defining iptable rules that remap these endpoint bindings transparent to the LoadLeveler code. In Figure 8(a), when CM needs to communicate to a StartD i $(1 \leq i \leq n)$, it sends the message to the respective destination IP address DIP_i but to a fixed port number LP_p. When this communication arrives at the StartD node, the packet is trapped by this iptable forwarding rule and forwards it to the appropriate port LP_i based on DIP_i. A similar rule coordinates communication originating from the SchedD to the StartDs.

StartDs are made to appear to the CM that they originate from unique IP addresses using the following three steps:

– The configuration file of each StartD is setup so that it uses a private port to communicate with the CM. Although CM is not actually listening on this port, this change is needed to identify which packets belong to which StartD at the iptables layer. When the StartD initiates a communication to the CM, its packets have this private port as the destination port in their header.
– Create an iptable rule that captures all outgoing packets from any StartD to the CM (DIP_{cm}), and based on the CM destination port number (LP_i) on the packet header, assigns a corresponding StartD IP address as the packet source IP address SIP_i, as shown in Figure 8(b). With this method, each outgoing packet to the CM is correctly labeled with the StartD that generated the packet but its destination port is not the actual port where CM is listening for StartD communications.
– Create an iptable rule on the CM node that redirects traffic destined to the list of private ports $(LP_1, LP_2, \ldots, LP_n)$ to the public port (LP_p) on which CM is listening.

The above methods create a large LoadLeveler cluster with minimal compute and memory resources. The iptables setup is automated with a the help of a few Perl and shell scripts. These scripts are parameterized so that the required number of StartDs may be activated on different physical machines. This setup is used to exercise the scheduling algorithms in the CM up to several thousand computes nodes.

A major limitation of iptables is their performance. Anecdotal evidence suggests that large numbers of iptables rules degrades the network performance

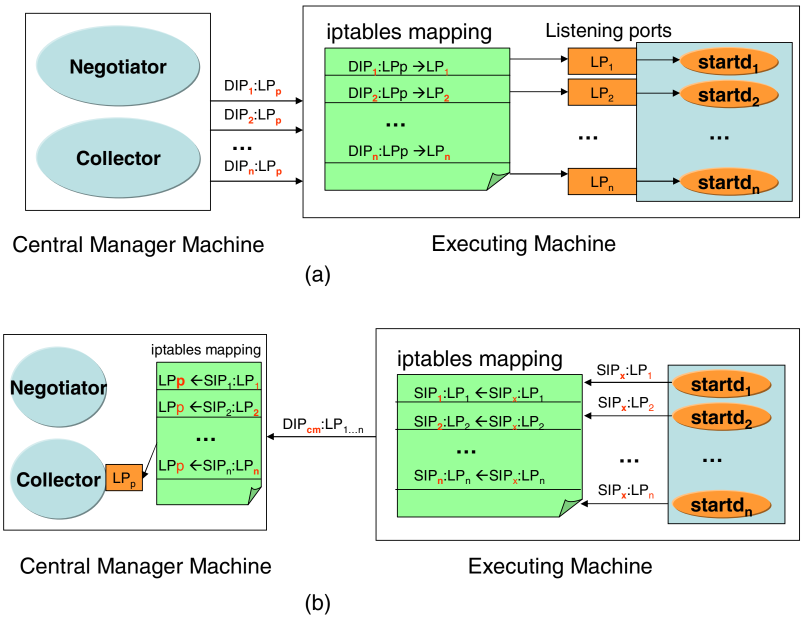

Fig. 8. LoadLeveler with virtualized StartDs – Iptables

because these rules are processed sequentially for every packet. Since network performance is a critical component of the hierarchical communication performance of LoadLeveler, the iptables approach is less favored than source code modifications described in the main text to orchestrate communications between CM, SchedD and StartDs.