Partitioned Parallel Job Scheduling for Extreme Scale Computing

David Brelsford, George Chochia, Nathan Falk, Kailash Marthi, Ravindra Sure
{brels4d,chochia,nfalk,kmarthi}@us.ibm.com, ravisure@in.ibm.com
IBM Systems and Technology Group

Norman Bobroff, Liana Fong, and Seetharami Seelam
{bobroff,llfong,sseelam}@us.ibm.com
IBM T. J. Watson Research Center

Abstract. Recent success in building petascale computing systems poses new challenges in job scheduling design to support cluster sizes that can execute up to two million concurrent tasks. We show that for these extreme scale clusters the resource demand at a centralized scheduler can exceed the capacity or limit the ability of the scheduler to perform well. This paper introduces *partitioned scheduling*, a hybrid centralized and distributed approach in which compute nodes are assigned to the job centrally, while local resource to task assignments are performed subsequently at the compute nodes. This reduces the memory and processing growth at the central scheduler, and improves the scaling behavior of scheduling time by enabling operations to be done in parallel at the job nodes. When local resource assignments must be distributed to all other job nodes the partitioned approach trades central processing for increased network communications. Thus, we introduce features that improve communications such as pipelining that leverage the presence of the high speed cluster network. The new system is evaluated for jobs with up to 50K tasks on clusters with 496 nodes and 128 tasks per node. The partitioned scheduling approach is demonstrated to reduce by an order of magnitude the processor and memory usage at the central processor. A similar gain is also observed in the time to schedule and dispatch jobs.

1 Introduction

The primary goal of job scheduling for high performance computing (HPC) is to assign parallel jobs to compute nodes, matching the resource requirements of the job to capable nodes. Traditionally, job to node matching is performed by a centralized scheduling architecture in which a resource manager module monitors the state of the compute nodes in the cluster, sharing this data with the collocated scheduling module. However, this centralized scheduling model is stretched in meeting the demands of petascale HPC supercomputer clusters..

For example, an upcoming supercomputer that is part of the IBM DARPA HPCS [1] program, contains up to 16K nodes and targets sustained performance

in excess of a petaflop for applications such as computational biology and chemistry, fluid mechanics, and galaxies formation studies. Each compute node consists of a quad Power7 chip module with 8 cores per chip for a total of 32 cores. Each core supports simultaneous multi-threading (SMT) up to 4, allowing up to 128 job tasks to execute on a compute node. Compute nodes are packaged in modular groups of 32 termed super-nodes. Internal to each super-node is a full point-to-point switched network with minimum sustained bandwidth of 6GB/s. The supercomputing system is constructed by inter-connecting tens to hundreds of super nodes with multiple 10GB/s links, up to a total of 16K nodes and capable of running 2 million tasks.

While a centralized scheduler scales linearly in memory and processor demand with the number of job nodes and tasks, it still may not deliver adequate scheduling performance and can be overloaded in demand for memory. In fact, we have found this to be the case while doing performance studies aimed at large scale systems using the IBM TWS-LoadLeveler (LL) scheduler [20, 4]. A root cause of the problem is that centralized schedulers do not leverage the large number of compute nodes assigned to the job that can assist in the scheduling process. Increasing the parallelism allows a transition of large parts of the scheduling and dispatching problem from linear scaling to more constant order, with substantially reduced resource demands at the central scheduler. This transition to a distributed and parallel model is enabled by the higher speed and lower latency networks that are now present on HPC clusters since distributed models invariably require more communication than centralized ones.

To this end we introduce a hybrid approach to the scheduling architecture which adopts key elements of both parallel and distributed system features that we call *partitioned scheduling*. The main observation that leads to partitioned scheduling is that by deferring the assignment of local compute node resources to tasks - such as DMA windows and processor affinity - until after the node selection process allows the compute nodes themselves to do scheduling of these local resources. Of course, certain local scheduling decisions such as network adapter assignments still have to be propagated to all other nodes in an all-gather type of communication paradigm [17]. A further contribution to the work is the many improvements in system communication developed to distribute the local node scheduling decisions and improve dispatching.

The new partitioned scheduling provides the greatest gain reported here. Once local resource scheduling is moved out of the central manager, additional improvement to the linear aspects of global job to nodes selection are possible by parallelization. Concurrently threads in node assignment function are leveraged using multicores while avoiding slowdowns arising from lock contention.

This paper presents the partitioned scheduling design and implementation and shows how it has led to significant speedups in job scheduling while simultaneously reducing the resource demand at the central manager. We show that even in small clusters of 248 nodes this trade-off in costs is considerable, leading to order of magnitude improvements in many performance aspects of job scheduling and dispatching. From an overall system evaluation perspective the

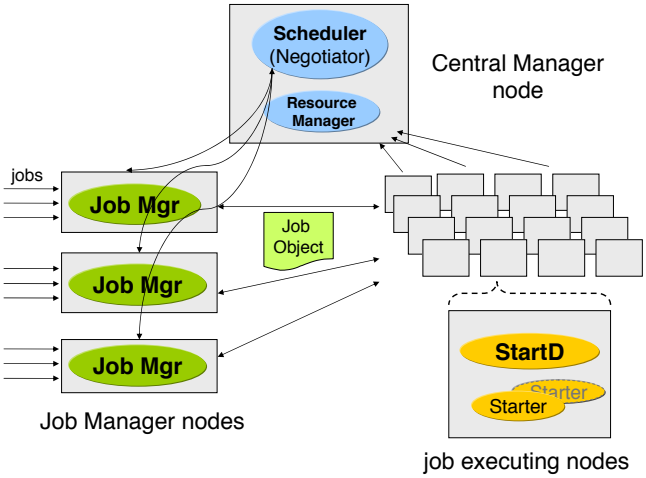


Fig. 1. A high level view of LoadLeveler Architecture

total gain does not quite reach a factor of 10, but exceeds a factor of five on our smaller cluster. But the improvements provide a capacity and scaling safety margin that is also expected to be significant in absorbing more load as the system scales up to 16K nodes.

The paper starts with a description of the centralized IBM LL architecture that existed prior to this study. After an analysis of its scaling limitations, the adoption of the partitioned scheduling and the numerous scalability enhancements are described. Measurements on up to 496 nodes and 128 tasks per node are used to illustrate the gains in of scalability that have been achieved.

2 LoadLeveler Architecture

IBM LoadLeveler (LL) is a job management product for high performance computing (HPC) clusters. Derived from the 1990's version of Condor [21] developed at the University of Wisconsin, LL focuses on batch job management for enterprise systems with commercially pertinent features, such as detailed accounting for charge-back, priority & work flow control, high availability with parallel check-pointing, and job recovery. It supports a wide range of systems from small clustered workstations to large IBM BlueGene [7] supercomputers.

The basic architecture of LL (Figure 1) consists of three key components: a central manager (CM), one or more job managers (JM), and cluster compute nodes. The JM provides a portal for job submission and is responsible for job life-cycle management including; persisting job state to disk, launching and coordinating the execution of the job on the nodes assigned by the scheduling component, and recovering the job in case of a failure. The CM runs on a single machine image and includes the collocated resource manager (RM) and job scheduler (aka Scheduler). The CM has a simple fail-over based recovery

model to standby CMs. In case of failure, job states are supplied by the JMs and resource states reestablished by compute nodes via the RM.

The RM is an in-memory repository of cluster information including the state of every node, including dynamic and static attributes. A daemon process (`StartD`) on each of the compute nodes pushes the resource information to the RM. `StartD` reports both the static and dynamic attributes of the node, the former at node start-up and reconfigurations, and the latter at configurable sampling intervals or job state changes. Static configuration data include hardware attributes, memory size, types and numbers of network adapters, while examples of dynamic metrics are active job state and CPU utilization.

The Scheduler allocates compute nodes and other resources to a job by matching the requirements of the job to compute nodes using the state data provided by the RM. In addition to number of nodes and number of tasks per node, the job specifies computational processor power, memory space, system architecture type (e.g., Intel x86, IBM POWER), disk storage, high-speed network resources, software environmental entities (e.g. OS, software license), etc. When available and supported in a clustered systems, jobs can specify a number of *network windows* that provide an efficient mechanism to exchange data between job tasks via direct memory access (DMA).

It is useful to characterize the resource types considered by the Scheduler into categories according to the job matching conditions:

1. **Global or floating resources**: A global resource is a cluster-wide consumable of finite number N such as cluster-wide software licenses for a particular application. No more than N jobs requiring such a resource can run concurrently on the cluster
2. **Node resources**: A node resource has static or/and dynamic attributes that are defined on a per node or per job basis, irrespective of the number of tasks of job running on that node. For example, a job may specify it needs to run in exclusive mode. Once a job of that type is assigned to a node, no other job can be assigned until that job releases the node.
3. **Node local resources**: Node local resource are a resources tied to a node. They are typically consumed on a per-task basis and the association between the task and the corresponding resource being consumed must be preserved for the duration of the job. Two common examples here are *task to core* mapping and *task to network window* mapping. In HPC, many times, a particular task runs on a particular core for the duration of the job using the *task to core* mapping. Similarly, in *task to network window* mapping, each task will be assigned a particular network window for its use. Every other task of this job must use this window to communicate with this task. For this, every task must know the task to network window association of all tasks of a job. This requires an all-to-all communication of this information.

The job-to-resource matching complexity is different for each category of resources. Global resources are matched on a per job basis while node local resources are assigned to individual tasks of each job. The complexity of matching

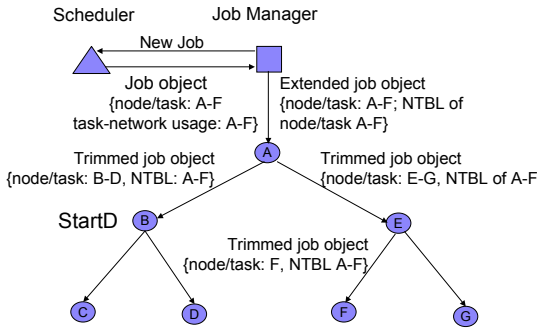


Fig. 2. Hierarchical Job and Network Usage & Table Propagation

leads to considerable processing, memory consumption, and a scalability bottleneck at the Scheduler. The subsequent section expands on the limits of processor and memory scalability at the CM machine.

After matching, the CM returns the resource assignments to the JM, including global, node and node local resources. If network window to task assignments are provided by CM for a job, then JM takes the per task network adapter information and builds it into the Network Table (NTBL) which contains sufficient information to enable all-to-all inter task communication. JM then merges the job node assignments with the NTBLs and wraps it into a single large job object (JO) which is sent to the `StartD` on the compute nodes. JO dispatching is done using the N-ary tree, where the structure of the tree is encoded into the JO itself. The JO is first passed to the root node where data for that node is extracted. The node then forwards the JO to its children and the process continues to the leaf nodes as in Figure 2 with a job scheduled for nodes A through E.

Job execution at compute nodes is managed by the local `StartD` which forks a `Starter` process – a job executing agent – to initiate and control the job execution. Concurrent execution of tasks at a compute node is enabled by forking multiple `Starter` processes. Changes of job status will be reported by `StartD`'s back to the corresponding JM that manages the job's lifecycle.

3 Scalability Issues in LL architecture

The centralized nature of job scheduling causes a processing and memory that is a concern for large scale clusters. CPU bottlenecks are examined first by looking at the scheduling stages, and identifying where they consume cycles that can be moved to the local nodes. Then we consider how the memory footprint, which becomes unsustainable, in a central architecture, is reduced by node level scheduling.

LL uses a two stage scheduling process in which a set of nodes *capable* of running the job (based on specific job attributes) is selected first, followed by a

priority based selection of a subset of these nodes according to *dynamic capacity*. These steps are further described below:

- Stage 1: Designate capable nodes as having the *static capabilities* to match the requirements of the job; capabilities include node architecture types and features, job class definitions, and high-speed network adapter types and counts.
- Stage 2: Select capable nodes from an ordered list of nodes that have the *dynamic capacities* to assign to the job; exemplary capacities include unused job class instances, unfilled multiprogramming level, and spare network windows. The nodes are ordered based on an administrator defined priority such as descending order of free CPU utilization.

Stage 1 is performed once for a job to produce a list of capable nodes. (An update is triggered only for the addition or removal of nodes from the cluster, an infrequent event in HPC environments.) Stage 2 is repeated at each scheduling cycle as existing jobs terminate and free up resources until enough nodes are available to dispatch the requesting job. The two stage scheduling process is an optimization that is useful if typically incoming jobs being scheduled must wait for termination of running jobs. In such cases, only stage 2 is repeated as needed.

The processing time to complete these stages is bi-linear in the number of job nodes and node local resources that need to be assigned to the job. In fact, node local resource scheduling time is proportional to the number of tasks because local resources are assigned on a per task basis. A typical example is allocation of network windows for DMA to each task on a node. Here the processing of stage 2 is logically an outer loop on the N nodes with an inner loop assigning network windows to T tasks. This CPU intensive work is quantified using the observed scheduling time (upper trace of Figure 8 in Section 6) which shows that scheduling a job of 248 nodes and 64 task per node takes 2.4 seconds on the centralized Scheduler. Assuming at best a linear scaling in nodes and tasks, this projects to at least 160 seconds on the 16K node cluster.

One could argue that, performed centrally, there is an opportunity to unwrap the scheduling loop, for example by having each of a group of hardware threads perform the local task scheduling for each node. However, a second major scaling issue arises which is that memory usage is becoming unmanageable. According to our data on a 248 node, 64 task job (upper trace of Figure 7 in Section 6), memory consumption is about 64MB. This scales out to over 3.6GB on the 16K node cluster. In fact, as long as the cluster is fully occupied this much memory must be consumed to keep track centrally of resource assignments of all currently running jobs. The scalability issue with memory is not simply the size, but the fact that it is not a monolithic allocation, but comprised of a very large number of smaller data objects reflecting the scheduling details. These allocations and memory releases add significant overhead to the CM.

The solution to the scalability problem is to leverage the compute nodes to perform the scheduling of local resources such as network windows. This reduces CPU utilization at the central manager by separating the scheduling into two

sequential steps. First, the Scheduler selects job nodes that have sufficient free local resources for the number of tasks per node. This is done by maintaining a simple count of, for example, each node's free network windows at the CM. Once the job nodes are selected, they assign the windows to node tasks and report these assignments to peer nodes that require communication. More detail of how this partitioned scheduling works in practice is described in the next section. The main point is that a large amount of the required processing of the Scheduler is now performed as a short serial process followed by a large parallel and distributed phase.

The distributed aspect of partitioned scheduling also solves the memory usage problem, dividing the required allocation among a large number of nodes. It improves other scale driven concerns which although less important, contribute in a measurable way to the overall system performance. For example, in the basic design of the prior section, the JO that is distributed to all nodes is very large and each node must read it into memory and scan it to find the data specific to that node. With a much smaller object optimizations in the processing and distribution of the JO transmission and handling are possible as described in Section 5.

Finally, we note that even the remaining sequential process of selecting the nodes which remains on the CM can be parallelized to leverage the multi-core hardware. The challenge in achieving a linear speedup in using a modest number of threads is to avoid locking conflicts on the per node data. Section 4.2 describes our multi-threading optimizations of these scheduling stages.

4 Partitioned Scheduling

The main observation that leads to a reduction in both processing and memory consumption in the Scheduler is that the node local resource assignment to each task can be performed in parallel by the compute nodes that are selected for the job. In order for the Scheduler to know job tasks can be dispatched to a node based on availability of sufficient node local resources, the RM only has to keep a count of total and free local resources for each node.

This motivates several new architectural aspects of LL, enhancing the base model of Section 2. The most significant is the partitioning of the scheduling function of per task local node resources to individual nodes. While not as broad in scope, the multi-threaded job scheduler improves scalability for large clusters and jobs. Finally, introduction of pipelining (5 to the distribution of JO and NTBLs has a significant impact on total schedule and dispatch timing.

4.1 Partitioned Scheduling of Local Resources

In the partitioned scheduling design, a scheduler agent (`Schd Agent`) is introduced to `StartD` to perform the detailed assignment of node local resources to tasks of job scheduled on the node, as shown in Figure 3. In the example of network window scheduling, only the static attributes of the network adapters

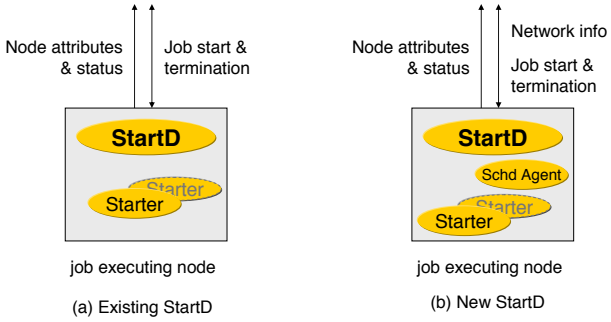


Fig. 3. Designs of StartD functions

and the count of available network windows are reported by `StartDs` to the RM. During the scheduling cycle, Scheduler matches only the available count of windows and the type of network adapters to the job request.

The majority of the node local resource scheduling assignments made by the Scheduler Agents are propagated to the JM to persist complete job information for failure recovery. Also, to enable all-to-all inter-task communication using network windows, these assignments need to be distributed to all other nodes.

In the new model the JO is first distributed to the centrally selected job nodes using the hierarchical tree of Figure 4. Upon receiving the JO, each node schedules local resources including network windows and starts the process of building the entries for the NTBL (Figure 4). Here, LL uses a variant of the well known *all_gather* implementation in which the unique information (adapter window) from each node is sent up the tree and consolidated at the root, after which the full set of information is distributed to all nodes through the same tree. This process starts from the leaf nodes where the assignments are passed to the parent and merged with those of its siblings. This continues up the tree as each node builds a partial NTBL from its own assignment and those of its child subtrees and so on up the tree, as shown in Figure 5. One optimization is that instead of passing all NTBL data to the root node, building a full NTBL and then sending the full NTBL down the tree to all nodes, each parent (including the root) sends the NTBL for each child subtree down the sibling subtrees. This reduces by half the amount of data exchanged to build the NTBLS for a binary tree.

The potential downside to the partitioning scheduling model is this necessity to communicate the detailed local scheduling decisions such as NTBLs to the all job task peers. If this distributed building of the NTBLS is significantly slower than the centralized in memory creation at the JM and subsequent distribution, then no net gain in scalability is achieved. In other words, based on network latency and bandwidth and processor speed at the CM, there is a cluster size at which performance will be better in central scheduling as opposed to this partitioning model. We have found with modern hardware that even a modest

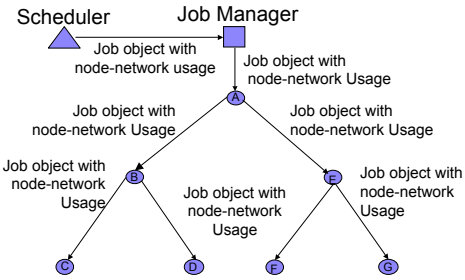


Fig. 4. Job & Network Usage Distribution

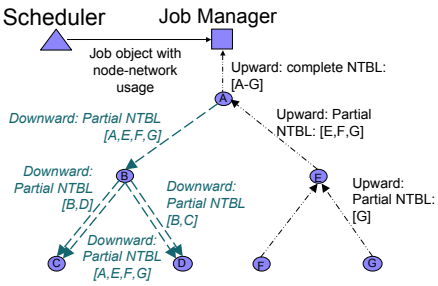


Fig. 5. NTBL Building & Distribution

cluster size of 248 nodes sees substantial gain from the changes, even as the number of tasks per node approaches unity.

4.2 Multithreading in Resource Matching

Scheduling is processor intensive and is redesigned here to leverage multicore and SMT capability using a master-worker design with multiple worker threads, as shown in Figure 6. The master thread assigns units of work, such as perform matching and selection from a block of nodes, and the workers pull the work from a queue. As each worker operates on a subset of the node list, it generates new data that must be reflected in global counters, for example, the number of software licenses consumed or the number of nodes successfully scheduled. These updates are handled by passing that data back to the master thread upon thread completion. The master is also responsible for error recovery and re-dispatching work from failed threads. This design ensures there are no potential access conflicts between threads to avoid the overhead of lock contention and lock recovery for failed threads.

The improvement of the multithreading design is borne out by the data showing the total time spent matching and selection process reduced from 14 seconds to 4 seconds on a four core system for 10K nodes and 32 tasks per node. All of the experiments reported in the evaluation section 6 include this optimization.

5 Pipelined Communication

Once a job is scheduled the job specific resource information for all tasks is encapsulated in the Job Object (JO) which is distributed to the assigned compute nodes using the n-ary tree of Figure 4. Each job has a unique tree which is dynamically constructed by from a structured list of nodes contained in the JO

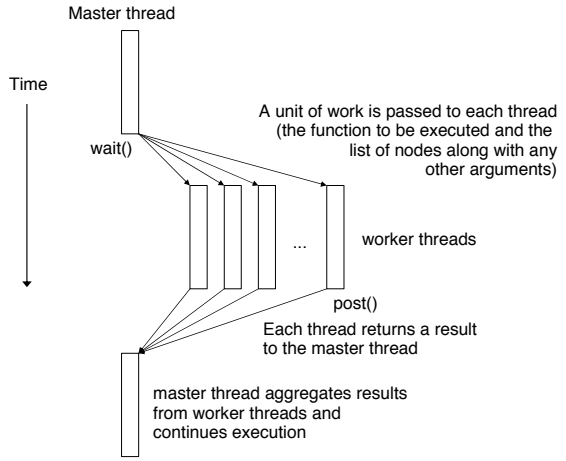


Fig. 6. Application of multi-threading to Scheduling

and organized by parents and children starting at the root. The tree is persistent for the job life-cycle and used for all subsequent LL communication for that job, both down and up the tree.

There are two modes of JO distribution down the tree; store-forward, and pipelined. In store and forward, a message is completely received at a parent node prior to forwarding to the children. The original design of the communication tree attempts to minimize data sent down the tree and always uses store and forward for messages with node dependence such as the JO. In this design the JO is opened at each node and new trimmed JOs are constructed specific to the sub-tree of each child, each trimmed JO being approximately half the size of its parent for a binary tree. In between the store and forward operations, a starter process is initiated to begin job launch on the node. Even though the starter is on its own thread, it is executing during the store and forward and is therefore on the critical path for messaging.

However, through performance analysis it is determined that the CPU time required for message trimming dominates the network transmission times [12]. The trimming is complicated as JO are XDR [2] encoded messages to support communication between heterogeneous systems. Encoding and decoding XDR is compute intensive. This is true even for fast CPUs and slow networks (100Mb Ethernet) and the gap between the processing bottleneck and network communication time widens with higher speed networks. Since processing time is the bottleneck, it is preferable to send larger messages to other nodes as fast as possible so they can begin message processing in parallel. Thus, a second communication mode, pipelining, is added. In pipelining the JO message is logically divided into 'chunks' and a node receiving data from its parent starts forwarding

to its children upon receipt of each chunk. The XDR processing is performed concurrently with the pipelining by a separate thread on the parent node without interference to the communication threads connected to the child nodes.

The immediate forwarding of partial message data in pipelining greatly reduces the latency, with leaf nodes beginning to receive the JO after approximately $log_2(N)$ chunks have been sent from the root to its immediate children. Pipelining achieves almost O(constant) scaling for a large message, where store and forward would be $log_2(N)$ times the total transmission time. This is already an order of magnitude improvement with a 1024 node job. PIpelining also benefits NTBL distribution where it is used to send the partial NTBLs from the tree root to the leaf nodes. More details of the performance gain and its overall contribution to the dispatching budget are given in Section 6.

6 Evaluation

In this section, we present our evaluation of the partitioned scheduling and optimizations discussed in the previous sections. We discuss the experimental setup, performance improvements in the memory growth at JM, scheduling time improvements at the CM, and the job dispatch time improvements.

We report on results from two versions of LoadLeveler: 1. a baseline version before our enhancements (V3.5.1.0), and 2. a new version with our design for partition scheduling and enhancements described in this paper (V4.1.0.1) [5]. We refer to V3.x as "Old" and V4.x as "New" design in the description below. We present the data from the old design to substantiate our motivations outlined in the introduction section. We juxtapose this with the data from new design – not so much to show the limitations of the previous design – but to highlight the potential improvements and the trends associated with the performance of the new design in job scheduling and dispatching operations on large scale systems.

6.1 Experimental Setup

The design and implementation work spanned over two years and in this time we had access to multiple clusters with different hardware and operating system configurations. We used different clusters for experimental validation of the different components of new design. In this paper, we report on the results from two representative large scale Power6 clusters: one 497-node cluster and another 249-node cluster.

The 497-node Power6 cluster is connected with Infiniband interconnection network. This cluster is used to obtain the experimental results for the memory usage analysis at the Job Manager, and the partitioned scheduling design that trades off some of the functionality from CM and JM to the compute nodes. Of the 497 nodes, one node is used for CM and JM and the remaining 496 nodes for job execution (compute nodes). The node with CM and JM consists of 8 Power6 cores and 32 GB memory, which provides sufficient memory space to study the memory usage with the old and new designs. Each of the 496 compute

nodes consists of 4 Power6 cores and 14 GB memory. So, the cluster consists of 1980 Power6 cores for job execution and 8 cores for LL operations. All 497 nodes consist of a single network adapter per node. All of these nodes are configured with AIX 6.1 and used exclusively for the performance evaluation.

The 249-node Power6 cluster also connected with InfiniBand interconnect is used to study the job dispatching optimizations. In this cluster, the CM and JM node consists of 8 cores and 32 GB memory and the remaining 248 compute nodes consist of 4 cores and 14 GB memory each. All 249 nodes are configured with SUSE Enterprise Linux Server (SLES) version 11.

6.2 Job Manager: Memory Consumption Data Capture and Analysis

In this section, we study the JM memory consumption for jobs requiring 496 nodes with different number of tasks per node: from 1 task per node to 128 tasks per node. We study memory consumption with different number of tasks per node because the amount of state information is directly proportional to the number of tasks assigned to a node. Applications that exploit hybrid MPI/OpenMP programming models tend to use different number MPI tasks per node depending on the MPI and OpenMP parallelization characteristic of the application. The 128 tasks per node is interesting because it corresponds to an extreme end execution model on Power7 node: typical Power7 compute nodes consists of 32 cores and with SMT=4, 128 hardware threads therefore a single Power7 node could potentially execute 128 MPI tasks one per hardware thread.

As we discussed before, the memory growth occurs at the JM as well as at the CM. At the CM, the memory is consumed not only for task scheduling but also for other activities such as RM which processes incoming traffic of status and utilization updates so measuring the memory consumption for task scheduling cleanly at CM is a bit tricky. However, the memory consumption at the JM is only due to task scheduling and there is no other memory consumption as long as the task queue is kept constant. So, since both CM and JM consume proportionate amounts of memory for a given job and since it is easier to isolate and measure memory consumption per job for JM process compared to CM process, we capture this data from JM process while scheduling jobs to the 496-node cluster.

For each test case the JM is restarted and initial memory usage by the JM process $M_{initial}$ is recorded before any jobs are submitted to the cluster. Then a job is submitted to JM, is sent to the CM for resource matching. After the resource matching it will be returned to JM at which point, the JM will dispatches the job. The memory usage at this point is recorded again for the JM process as M_{final}. The difference between these two memory usages: $M_{job} = M_{final} - M_{initial}$ is the memory consumption by the JM process associated with a single job. These tests are repeated for LL old and new designs for comparison.

Figure 7 shows the JM memory usage for a 496-node job with increasing number of tasks per node from 1 to 128. For the old design, the data is captured for up to 64 tasks per node. From the data, it is obvious to see that the new

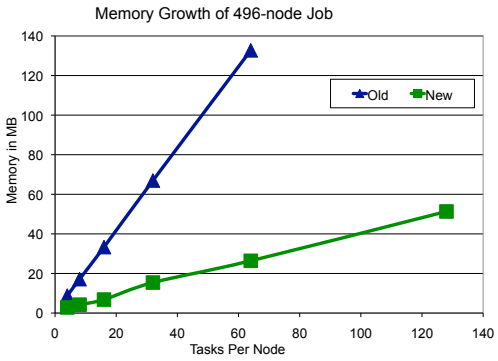


Fig. 7. Job Manager Memory Growth

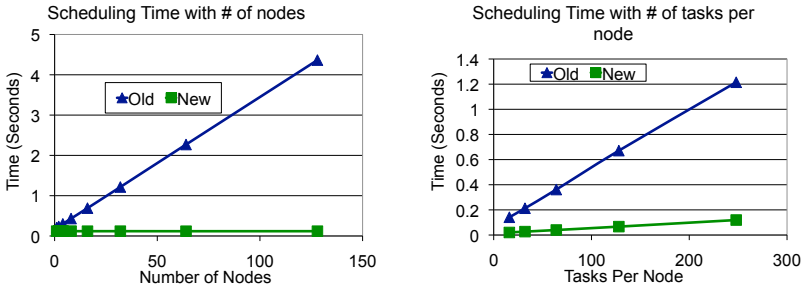
design results in substantial memory saving at the JM node – in fact about 5.5x lower memory usage in the new design compared to the old design. Recall that this cluster had a single network adapter while HPC clusters can typically have two or more adapters and the memory consumption in the old design increases linearly with the number of adapters while it increases only by a constant with the new design. The memory savings will be even more substantial for systems with multiple network adapters.

Now, what would be the JM memory for original and new design for the largest scale Power7 HPC system with 16K nodes? A linear projection of the data from Figure 7 from 496 nodes, 64 tasks per node to 16K nodes give us about 4.6 GB per job with old design and 0.83 GB per job with the new design. Keep in mind that job scheduling not only depends the memory usage but also on memory allocation/de-allocation time.

6.3 Central Manager: Job Scheduling Time Capture and Analysis

Job scheduling time is defined as the time between when a job request arrives at the CM from JM and when it is ready to be sent back to the JM with the task resource assignments. In this time, eligible resources are identified, allocated, and the job object is created with the assigned resource information in the Scheduler. The scheduling time does not include the job dispatching time, which is the time between when the job arrived at JM and when the JM gets notification from all compute nodes that they ready to execute the job. All evaluation measurements from our experiments were taken after the clustered system initialized to the idle state, i.e., they system is up and operational but it has no workload.

Figure 8(A) shows the scheduling time for jobs requiring different number of nodes with 32 tasks per node. Significant improvement in scheduling time is noticed for jobs running under the LL new design. There are two contributing factors to the improvement. First, the Scheduler is relieved from matching node local resources to individual task of the job, as described in Section 4.1. Second,



(A) (B)

Fig. 8. Partitioned scheduling performance evaluation. (A) Scheduler scheduling time per job with different number of nodes. Each job requests resources for 32 tasks per node. (B) Scheduler scheduling time for a 248-node job with different number of tasks.

the Scheduler is exploiting the multi-threading enhancements on Power6, as described in Section 4.2.

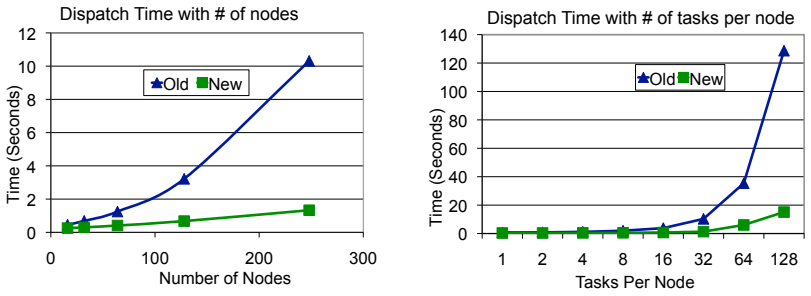
Figure 8(B) shows the scheduling times for 248-node job with varying number of tasks per node from 2 to 128. In the old design, because task specific resources are allocated at the Scheduler, the scheduling time increases linearly with the number of tasks per node. In the new design, this time becomes a constant because this task specific matching is offloaded from the Scheduler to the compute nodes. The data here further confirms this advantage of relieving Scheduler from matching node discrete resources to job tasks such that the scheduling time is independent with respect to number of tasks per node in the LL new design, as comparing to the increasing in scheduling time with increase number of tasks per node in the original design.

Data in Figures 8(A) and 8(B) together shows that the centralized scheduling performance can be improved significantly by carefully partitioning the resource matching between the Scheduler and the compute nodes. Thus partitioned scheduling can achieve performance and scalability while avoiding scheduling complexity and load balance issues associated with distributed scheduling.

6.4 Job Dispatching Time Analysis

Our partitioned scheduling moves the network table construction from JM to compute nodes. Network tables are constructed among the compute nodes by exchanging the node specific information as discussed in Section 4.1. Communicating node specific information requires exchanges between job nodes to disseminate the local adapter window assignments to all other nodes. In this section, we will analyze the job dispatching time with the old and new designs.

The job dispatching time for the new design includes: 1. the time to send job object from JM to the job to all compute nodes, 2. the time the for Scheduler Agents allocating network resources to individual local tasks, 3. the time to



(A) (B)

Fig. 9. A. Job Dispatch time as a function of number of nodes. B. Job Dispatch Time as a function of number of tasks node of a 248 node job

building partial Network Tables for local tasks, 4. the time to hierarchically build the partial network tables and distribute them to all nodes, and 5. the time to start the MPI master task.

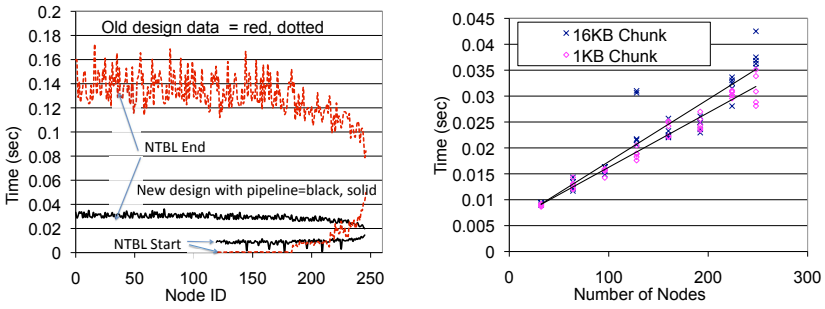
Similarly the job dispatching time for the old design only includes steps 1 and 5 of the new design, i.e., the time to send job object including network tables from JM to the compute nodes, and the time to start the master task.

Figure 9(A) shows the job dispatching time as a function of number of nodes requested by the job for the old and new designs. In this example, the job requests 32 tasks per node. As shown in the figure, the dispatch time for old design grows approximately as a quadratic function where as for the new design it is a linear function of the number of nodes. For a 248 node job, the job dispatching takes more than 10 seconds in the old design while it takes under 2 seconds with the new design, resulting in more than 5x improvement. Similarly, Figure 9(B) shows the job dispatching times for a 248 node job that requests different tasks per node with the old and new designs. In this case, for a 248 node job with 128 tasks per node, the job dispatch time improves from 130 seconds to under 20 seconds, resulting in a more than 6x improvement.

These substantial improvements in the new design – despite the extra steps associated for the network table construction and dispatching – are because it efficiently exploits the parallelism in allocating and distributing network resources by the agent schedulers at the compute nodes.

7 Job Dispatch Optimization: Pipelining

Even though the new design achieved over 5X speed-ups in job dispatching time and fundamentally altered the quadratic function to a linear function, the dispatch time is still deemed excessive, especially as we project it to a 16K node system with 32 to 128 tasks per node. To further optimize the dispatch time we implemented a set of optimizations, some just restructuring the existing code



(A) (B)

Fig. 10. A. Communication Performance for Network Tables. B. Pipeline tuning for NTBL distribution, no data size scaling

and others include efficient exploitation of system resources. We report on the performance of the pipelining optimization in dispatching the job object and in dispatching the network tables with the new design. In the figures below the "new" refers to the partitioned scheduling design which uses the store-forward method and the "pipelined" refers to the optimized partitioned scheduling with pipelining.

7.1 Impact of Pipelining on Network Table Dispatch

Figure 10(A) shows the start and completion of Network table (NTBL) construction on a per node basis for a binary tree. The horizontal axis is the node number where the nodes are ordered by level in the tree with the leaf nodes first, then the level above the leafs, and so on. Since the tree has 248 nodes, nodes 1 thru 120 are leaf nodes, while the immediate children of the root are at the far right, and the root is not shown. The vertical axis is the time that the NTBL is received at that node. The new design data set is colored as black and in solid line, while the old design data is in color red and in dotted line.

Since partial tables are built on the way up the tree and the full NTBL is sent down the tree, each data set contains two lines corresponding to the direction of data flow: 1) A lower line for the time a non-leaf nodes receives the partial tables from its children and 2) An upper line (later time) when that node receives the combined network tables for the other sub-trees from the root as they are passed back down the tree. Since leaf nodes don't receive partial NTBLs on the way up, they only have one value and do not appear on the lower trace of each data set. The gap at the far right measures the time it takes for the immediate children of the root to forward the NTBLs for their subtree to the root, and for the root to turn around and start passing the NTBL for the down complementary sibling's subtree. Thus, time on the upper curve is when that node receives the complete NTBL and the latest time on this trace marks completion of the distribution of

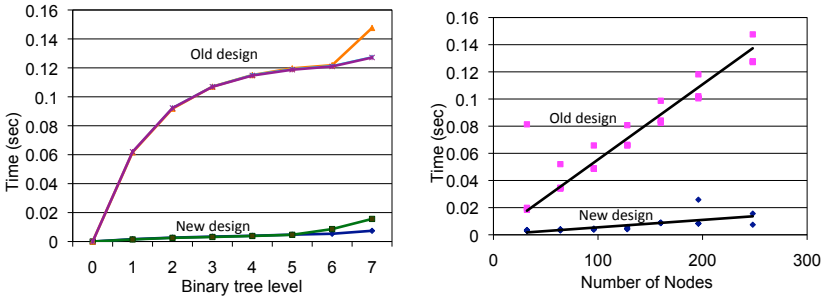


Fig. 11. A. JO distribution with depth of tree, B. JO distribution with number of nodes

NTBLs. This type of per node data chart is used by us to analyze performance as in addition to general trends it shows variability in the communication chain.

The new NTBL data design completes about 5 times faster because of pipelining. The coefficient of variation in the upper trace is also reduced indicating better predictability and stability in the new system, an important consideration in scaling to 16K nodes.

Detailed performance data for pipelined distribution of NTBLs is provided in Figure 10(B). It shows the time to complete NTBL construction and distribution on the 248 node scale test cluster with different chunk sizes in the pipelining process. There are 32 tasks per node so the NTBL size is of the order of 200KB. The data series labeled 1KB and 16KB reflect the chunk size of the pipeline, i.e., the size of the message received at a node before starting to forward. The effect of the chunk size is noticeable as the scaling is much better with 1KB chunk size as expected by the design, small chunk size benefits the pipelining design. However, keep in mind that smaller chunks increase number of communication steps so there is a trade off in the chunk size selection that must be considered carefully depending on the job size in terms of the number of nodes and the number tasks per node.

7.2 Impact of Pipeline on Job Object Distribution Performance

Figures 11(A) and 11(B) show the JO distribution using store-forward and pipeline schemes. In Figure 11(A) the horizontal axis is the depth of the binary tree used in the 248 node test cluster. The vertical axis is the time for the JO to be received at that node. In the data series labeled old design the store-forward communication mode is used. The inter-arrival time between each tree level decreases, as depth increases because the JO is trimmed by half and takes less time to process. Here the JO is about 200KB, and LL can drive the InfiniBand network at about 100MB/s to 200MB/s so the raw network time is around 1-2ms. Figure 11(B) compares the two communication modes as a function of the

size of the parallel job. The new design curve uses the pipelining and provides a nearly linear result, showing a greatly improved scaling in comparison with the original design of store-forward and achieving almost 7x improvement in JO dispatching.

8 Related Work

Decentralized or distributed scheduling has been proposed and evaluated for a long time [15]. [16] used the hierarchical scheduling as a way to aggregated resources from lower level to higher level in making scheduling decision. [14] provided a comprehensive taxonomy of distributed scheduling and touched upon the cooperative peer schedulers. However, the concept of partitioned scheduling and the way it is used in the paper has not been proposed to the best of our knowledge. The partitioned scheduling is a cooperative scheduling of node discrete resources after the centralized scheduling of node to a job is done.

Frachtenberg et al. [18] present Flexible Coscheduling that classifies processes based on their communication and computation requirements to increase system utilization by improving the job compaction in gang scheduled systems. It is a hybrid scheduling algorithm in the sense that there is a global scheduler for the system and a local scheduler per node. The local scheduler makes decisions on when to run a local task beyond its allocated time slot. In contrast, the LoadLeveler local scheduler schedules local resources for the task and alleviates the burden from the global scheduler to improve scalability of the scheduling system.

The terms partitioned scheduling [10] or semi-partitioned scheduling [6, 19] are used in real-time scheduling community, in contrast to global scheduling, for scheduling job tasks in multiprocessing systems. More specifically, the author's *partitioned scheduling* refers to having job tasks assigned to specific processors with a system, and then executed on those processors without migrations. The use of partitioned scheduling in real-time discipline is similar in concept to distributed scheduling [9] as the parallel job community. Our use of partitioned scheduling is different, and we refer to the scheduling assignments of node discrete resources to job tasks.

The paper of Bobroff [12] presented a lightweight virtualization approach of LoadLeveler and applied to study the scalability of job scheduling and dispatching in large scale parallel systems using a modest number of physical nodes. The results provided insights on scalability issues and inspired the re-design of LoadLeveler product.

Balaji et al. [11] and Butler et al. [13] describe API's for scalable process management in large scale systems. IBM systems have similar technologies and come with proprietary resource management software to exploit these hardware features. The LoadLeveler process manager coordinates the fine grained resource scheduling such as DMA allocations with the underlying IBM hardware platform resource manager. Layers such as IBM Parallel Operating Environment [3] provide the additional mechanisms for process boot strapping.

Although performance data of the old version of LoadLeveler showed significant scalability problems, it is used on several large scale systems such IBM BlueGene without these problems. Part of the reason for this is that the LoadLeveler design on IBM BlueGene works differently from how it works on general purpose large scale systems. For example, there are no LoadLeveler components that actually run on every node of the BlueGene system, which means it does not have to scale with the number of nodes of the system (see e.g., [8]). More importantly, each node of the DARPA HPCS system consists of many more resources compared to nodes of current generation HPC systems which cause the increased complexity and result in scalability issues for LoadLeveler. Our solutions in this paper address this complexity by carefully dividing the scheduling responsibilities between the global and the local scheduler.

9 Concluding Remarks

This paper presents significant enhancements to IBM LoadLeveler to achieve performance and scalability objectives in job scheduling for extreme scale computing systems. A set of architectural changes are introduced that bring distributed scheduling concepts to LL while preserving the benefits of centralized scheduling.

The most important of these is *partitioned scheduling* which performs task to local node resource matching at the job nodes instead of the central scheduler. This reduces the processing and memory footprint at the CM, leverages the ability to use the job nodes in parallel to assist with scheduling, and changes the scaling of scheduling time with tasks per node n from O(n) to approximately O(constant). This outcome is demonstrated by experiments with up to 50K tasks (128 tasks per node) for local scheduling of adapter window assignments, a particularly challenging case because the local scheduling results have to be propagated to the other job tasks to enable all-to-all communication.

Closely related is the observation that trading off an increase in total processing done in parallel at the job nodes, for more network data transmission is often a favorable one with modern high speed, low latency networks. This is taken advantage of in the dispatching phase by sending the full JO to every job node so they can start processing on the JO concurrently. In looking to further reduce network time and increase concurrent processing during the JO and NTBL distribution, the underlying LL communication algorithm was reexamined leading to a change from store and forward to pipelining.

Finally, the scheduling phase of the central manager is architected to take advantage of the underlying multicore and multithreaded hardware by avoiding lock contention. This contribution reduced the CM scheduling time from 14 second to 4 second on a four core compared to single core system achieving about the ideal 4X gain for a job with 10 K nodes and 32 task per node.

This paper presented data from relatively smaller clusters compared to our target system. In addition to the measurement we have used modeling and projections to reason about the scalability at target system scale. Based on this, we

expect the efficiency gain in the new design would be even more significant than it is shown in this paper. However, this will need validation with real data, and that is a key part of our future work.

References

1. DARPA High Productivity Computing Systems project; http://www.darpa.mil/IPTO/programs/hpcs/hpcs.asp.
2. External Data Represenation Standard; http://tools.ietf.org/html/rfc1014.
3. IBM Parallel Environment (PE); http://www-03.ibm.com/systems/software/parallel/index.html.
4. IBM Tivoli Workload Scheduler LoadLeveler; http://publib.boulder.ibm.com/-infocenter/clresctr/vxrx/index.jsp.
5. IBM Tivoli Workload Scheduler LoadLeveler Version 4.1; http://www-01.ibm.com/common/ssi/rep_ca/5/897/ENUS210-145/ENUS210-145.PDF.
6. Anderson, J. H., Bud, V., and Devi, U. C. An edf-based scheduling algorithm for multiprocessor soft real-time systems. In *ECRTS* (2005).
7. Aridor, Y., Domany, T., and et. al. Open job management architecture for the Blue Gene/L supercomputer. In *Job Scheduling Strategies for Parallel Processing*. Springer Verlag, 2005. Lect. Notes Comput. Sci. vol. 3834.
8. Aridor, Y., Domany, T., and et. al. Open job management architecture for the blue gene/l supercomputer. In *JSSPP* (2005).
9. Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., and Culler, D. E. Effective distributed scheduling of parallel workloads. In *SIGMETRICS* (1996), pp. 25–36.
10. Baker, T. P. A comparison of global and partitioned edf schedulability tests for multiprocessors. In *Proceeding of International Conf. on Real-Time and Network Systems* (2005).
11. Balaji, P., Buntinas, D., and et. al. PMI: A Scalable Parallel Process-Management Interface for Extreme-Scale Systems. In *EuroMPI* (2010).
12. Bobroff, N., Coppinger, R., and et. al. Scalability analysis of job scheduling using virtual nodes. In *JSSPP* (2009).
13. Butler, R., Gropp, W., and Lusk, E. A scalable process-management environment for parallel programs. In *In Euro PVM/MPI* (2000), Springer-Verlag.
14. Casavant, T. L., and Kuhl, J. G. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Trans. Software Eng. 14*, 2 (1988).
15. Casey, L. M. Decentralised scheduling. *Australian Computer Journal 13*, 2 (1981).
16. Chandra, A., and Shenoy, P. J. Hierarchical scheduling for symmetric multiprocessors. *IEEE Trans. Parallel Distrib. Syst. 19*, 3 (2008).
17. Demaine, E. D., Foster, I. T., and et. al. Generalized communicators in the message passing interface. *IEEE Trans. Parallel Distrib. Syst. 12*, 6 (2001).
18. Frachtenberg, E., Feitelson, D. G., and et. al. Adaptive parallel job scheduling with flexible coscheduling. *IEEE Trans. Parallel & Distributed Syst 16* (2005).
19. Kato, S., Yamasaki, N., and Ishikawa, Y. Semi-partitioned scheduling of sporadic task systems on multiprocessors. In *ECRTS* (2009).
20. Prenneis, A. Loadleveler: Workload management for parallel and distributed computing environments. In *Super Computing Europe (SUPEREU)* (1996).
21. Tannenbaum, T., Wright, D., and et. al. Condor – a distributed job scheduler. In *Beowulf Cluster Computing with Linux*, T. Sterling, Ed. MIT Press, October 2001.