High Throughput Grid Computing with an IBM Blue Gene/L

Jason Cope [#1], Michael Oberg [*2], Henry M. Tufo [*,#3], Theron Voran [#4], Matthew Woitaszek [*5]

[#]*Department of Computer Science, University of Colorado*
UCB 430, Boulder CO 80309, USA
[1]jason.cope@colorado.edu
[3]tufo@cs.colorado.edu
[4]theron.voran@colorado.edu

[*]*National Center for Atmospheric Research*
1850 Table Mesa Drive, Boulder CO 80305, USA
[2]oberg@ucar.edu
[5]mattheww@ucar.edu

***Abstract*— While much high-performance computing is performed using massively parallel MPI applications, many workflows execute jobs with a mix of processor counts. At the extreme end of the scale, some workloads consist of large quantities of single-processor jobs. These types of workflows lead to inefficient usage of massively parallel architectures such as the IBM Blue Gene/L (BG/L) because of allocation constraints forced by its unique system design. Recently, IBM introduced the ability to schedule individual processors on BG/L – a feature named High Throughput Computing (HTC) – creating an opportunity to exploit the system's power efficiency for other classes of computing.**

In this paper, we present a Grid-enabled interface supporting HTC on BG/L. This interface accepts single-processor tasks using Globus GRAM, aggregates HTC tasks into BG/L partitions, and requests partition execution using the underlying system scheduler. By separating HTC task aggregation from scheduling, we provide the ability for workflows constructed using standard Grid middleware to run both parallel and serial jobs on the BG/L. We examine the startup latency and performance of running large quantities of HTC jobs. Finally, we deploy Daymet, a component of a coupled climate model, on a BG/L system using our HTC interface.

I. Introduction

Integrated, low power systems such as the IBM Blue Gene family of machines are becoming quite popular. Reasons for this include maximizing aggregate flops per watt, specialized interconnects, and extreme scalability. However, the current offering in the Blue Gene family, the IBM Blue Gene/L, introduces its share of quirks when dealing with programs that were written for general-purpose clusters with the standard Linux software stack. Some of the challenges with the BG/L platform include a low amount of memory per compute node, different compute and compile architectures, and node allocation constraints.

Scientific modeling codes that are executed as complex workflows are troublesome when ported to systems such as BG/L. These workflows execute in discrete stages that may have various dependencies and consist of a mix of parallel and serial programs. Some workflows run multiple applications to produce a single final output, while other workflows run the same application multiple times with different parameters or repeat the same task over large data collections. For these workflows, different stages may be executed on different systems or even on different types of nodes within the same system.

The problem with executing these workflows on architectures such as BG/L is one of allocation constraints. Until recently, a single node on a BG/L system was not able to be allocated for a discrete task. Instead, processors were allocated by partitions, and the entire partition was capable of running only a single MPI application. If a task did not require all of the nodes in the partition, then the remaining nodes sat idle. This wastes compute time on the idle nodes and is antithetical to the power-efficient philosophy of such scalable architectures.

IBM recently extended their BG/L system software to provide a computing mode known as High Throughput Computing (HTC) where separate serial tasks can be executed on individual nodes of a machine. The remainder of this paper explores how best to exploit this new capability. Section II presents IBM's HTC mode and how it is implemented on the BG/L platform. Section III describes related work in the area of Grid workflows and the use of HTC on BG/L. Section IV describes the design of our high-throughput computing execution system, including integration with common Grid tools. We measure the performance of our task assignment system in section V, and in Section VI we test our system using real applications from an NCAR workflow.

II. Background

The IBM BG/L is a capability system that is designed for extreme scalability, emphasizing the execution of massively parallel MPI programs. Several specialized interconnects and a very energy-efficient design lead to scalability exceeding 128,000 CPU cores with power consumption typically 2-10x lower than commodity cluster systems [1]. With energy costs and heat dissipation requirements becoming limiting factors to

data center operation, the BG/L platform is becoming desirable for use with other modes of computing as well.

In early 2007, IBM released a technical preview of HTC support for BG/L. On BG/L, the smallest allocatable partition size is that of a nodecard, which contains 32 compute nodes and typically 1-2 I/O nodes. Because each node contains two CPU cores, the smallest partition that can be scheduled contains 64 CPUs. Although users can run smaller MPI jobs, or even a single-processor job compiled with MPI, allocating 64 CPUs when only one is desired is a waste of computational resources. With the introduction of HTC, MPI support can be forfeited to gain the ability to place independent tasks on each CPU in a partition.

IBM's suggested design for HTC utilizes a client-server model [2]. A partition is booted as usual, but instead of a parallel program, a launcher program is executed on each CPU in every node. Each launcher program contacts a central queue server and requests a task assignment. The specified task is spawned and executed. Once the task completes, the launcher program is reloaded and another task is requested. The process repeats until the partition is deallocated or no tasks remain.

III. Related Work

The primary areas of related work to the research we present are high-throughput computing and cluster virtualization. The most well known distributed high-throughput computing tool is Condor [3], [4], [5]. Condor provides tools for integrating a collection of resources, such as large quantities of idle workstations, into a unified resource pool with a batch style user interface. The goal of Condor is to better enable compute intensive workloads and it provides many additional tools to satisfy these types of workload requirements.

Additional Condor tools that are related to our work include Condor-G, DAGMan, and the classified advertisement (ClassAds) matchmaking framework. Condor-G [6] provides a Grid-enabled interface for Condor to integrate with Globus Grid-enabled resources. Condor-G extends the standard Condor user interface to Grid-enabled resources that use the Grid Resource Allocation Manager (GRAM) and stages files between Grid resources using GridFTP. DAGMan provides workflow management capabilities to Condor. Through DAGMan, users can define workflows as directed acyclic graphs (DAGs) and submit these job definitions to Condor through a special interface. DAGMan will parse the workflow definition and submit jobs to the specified Condor pool according to the rules in the workflow definition. Finally, the ClassAd feature of Condor allows users to define resource description parameters for their jobs [7]. Condor uses matchmaking tools to identify the appropriate resources available in the specified pool and selects the resources that meet the mandatory and subjective requirements as described in the ClassAd. Together, these tools can help integrate HTC resources into existing Grid computing environments. Our implementation, which utilizes a Globus GRAM adapter to interact with an HTC resource manager, can leverage the Condor–G, DAGMan, and ClassAd tools to support Grid-enabled workflows and workflow planning tools.

Recently, Condor developers and IBM investigated the use and deployment of Condor on the BG/L platform [8]. Their work produced a preliminary port of Condor for BG/L with HTC support. Though there is some similarity between our work and the Condor on BG/L project, our design philosophy and approach are fundamentally different. The Condor approach requires the deployment of Condor scheduling tools on the BG/L front-end nodes and includes the integration of Condor daemons into the BG/L I/O nodes. In contrast, our implementation aims to integrate with the existing local BG/L resource manager and batch queue system. Our implementation does not require changes to the BG/L I/O node system images to support scheduling daemons and utilizes the existing site scheduler for the allocation of resources. This approach simplifies the transition required to support HTC on an existing BG/L system and allows for scheduling both parallel and HTC based tasks.

Our work is also related to cluster virtualization. Virtual cluster technology provides tools for users or applications to create customized sub-clusters from partitions of much larger clusters. MyCluster [9] provides the tools and techniques required by Grid users to create a virtual cluster from a collection of Grid resources. These tools include resource allocators and integrators. MyCluster uses common Grid computing tools including the Globus Toolkit and Condor to integrate resources. While recent research with MyCluster has focused on strategies for allocating resources across a Grid, creating a stable virtual cluster from these resources, and adapting the proxy and resource allocation mechanism to throughput requirements, our work focuses on partitioning resources from a local BG/L for use in a Grid environment or a virtual cluster. We envision that the exposed Grid interface for HTC could be integrated with higher-level tools, such as MyCluster, for creating virtual clusters in Grid computing environments.

IV. Design and Implementation

Our high-throughput computing execution system accepts individual tasks using a standard Grid interface, aggregates tasks into partition-sized jobs that are run using the existing site scheduler, and employs a client-server architecture to assign tasks to computational nodes (See Figure 1). Grid clients, such as DAGMan, Condor-G [6], and other Grid-based workflow systems submit tasks for execution using the standard GRAM interface. The HTC GRAM service passes incoming work requests to the HTC partition controller. The controller creates and submits jobs to the local batch queue scheduler – in our case, the Cobalt scheduler [10]. When Cobalt executes the job, each node on the partition runs the HTC launcher, which sends a request for work to the dispatcher component of the controller. The dispatcher replies with a task assignment for the node. When the task completes, the laucher is reloaded, and another task is requested. The process repeats until all HTC tasks have completed, or insufficient walltime remains for the partition to complete pending tasks.

The primary advantage to this design is that it supplements, rather than replaces, the existing site scheduler. A workflow

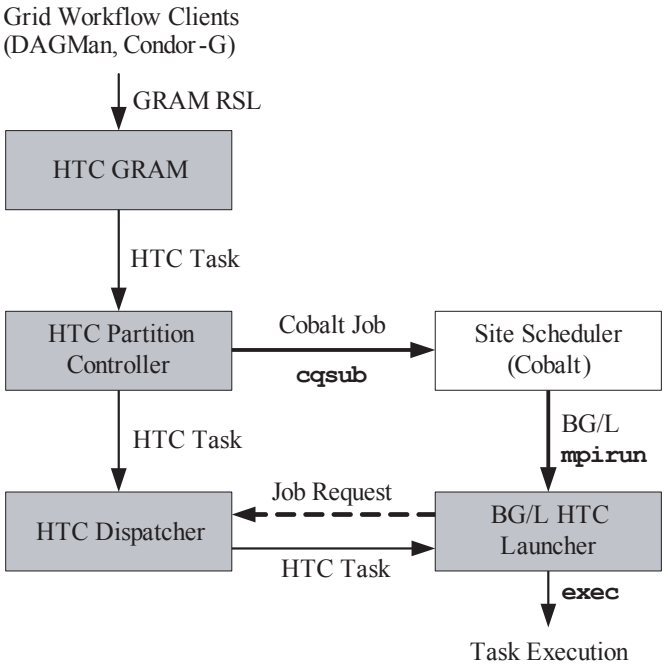

Fig. 1. High-throughput computing (HTC) task execution flow.

including both parallel MPI executables, as well as single-processor tasks, may run on a single BG/L system at the same time. Because the HTC tasks are aggregated and presented to the site scheduler as partition-sized jobs, the existing queueing and fairshare policies may still be used to mediate resource allocation. The entire machine may be used for parallel jobs, HTC tasks, or a mix of both depending on the workloads submitted to the system by the users with little or no advance configuration or intervention by the system administrators.

A. Globus Grid Integration

To provide Grid integration and interoperability, we developed an HTC scheduler adapter for GRAM. This adapter provides users and workflow management tools an interface to HTC through the standard Globus Toolkit tools, such as globus-job-run and globusrun. The HTC Grid integration toolset is composed of four components: HTC-GRAM Scheduler Adapter, HTC Scheduler Event Generator (SEG), HTC Web service, and HTC information provider. The bulk of the integration work is provided by the HTC-GRAM Scheduler Adapter and the HTC-GRAM SEG. The HTC-GRAM Scheduler Adapter provides an interface for WS-GRAM and PREWS-GRAM jobs to submit and cancel tasks and a monitoring interface for PREWS-GRAM jobs. The interface interacts with a database and builds the appropriate SQL commands to register tasks or monitor a task's state. The HTC-GRAM SEG provides a similar monitoring interface for WS-GRAM jobs and interacts directly with the database independent of the HTC-GRAM Scheduler adapter.

The HTC Web service adapter registers the HTC scheduler as a supported local resource manager for WS-GRAM. The HTC information provider calculates statistics on the resources managed by the local HTC resource manager. The information provider interacts with the HTC database and determines the current number of resources managed by HTC and the utilization of these resources. This information is also registered with the Globus Monitoring and Discovery System (MDS).

B. HTC Partition Controller

Even though the BG/L HTC implementation allows individual CPUs to execute independent jobs, the system's allocation mechanism continues to function at the granularity of partitions. Each partition consists of a number of computational nodes and at least one I/O node. Because an I/O node is allocated to a single user, all I/O is mediated through the I/O node, and the compute node kernel does not support `setuid`, an HTC partition must be booted for a specific user, and only tasks from that user may be assigned to that partition.

Instead of managing BG/L partitions directly, we allocate partitions dynamically using command-line calls to the underlying Cobalt site scheduler. (We also use a small collection of direct RPC calls to the Cobalt queue manager, but these could be replaced by parsing output from the queue status command of any scheduler.) As users submit HTC tasks to the system, Cobalt jobs are submitted automatically by the controller to start HTC partitions to run those tasks. Over time, as the walltime remaining on the Cobalt jobs becomes insufficient for further tasks, the controller automatically deletes idle Cobalt jobs and schedules replacements.

The HTC partition controller examines the queue of waiting HTC tasks and aggregates tasks with similar properties into logical task execution groups. The three task properties used for grouping are:

- User name
- BG/L node mode (coprocessor or virtual node)
- Accounting project number

These properties are set at the partition level, so only tasks with the same characteristics may be placed on the same partition. For each group of similar HTC tasks, the aggregator calculates the desired number of partitions that should be assigned using a heuristic that considers the total number of tasks waiting as well as the total walltime. New tasks are placed on scheduled or running partitions if they will fit within the remaining walltime, otherwise a new partition is requested from the Cobalt scheduler.

The partition controller does not employ any scheduling logic; rather, it merely creates Cobalt jobs that are scheduled by Cobalt itself. The number of Cobalt jobs submitted is proportional to the amount of work submitted to the HTC queue by the user, but the presence of queued jobs does not guarantee that the user will be allowed to use that much of the system. The underlying Cobalt queue policies and reservations determine how many partitions will be booted to run HTC jobs for each user. From the perspective of the Cobalt scheduler, an HTC partition request is simply another single-partition 64-processor job, indistinguishable from other parallel jobs in the queues.

C. Job Launcher and Dispatcher

When a BG/L partition is allocated in HTC mode, a launcher program is specified to manage the execution of tasks on each node. This launcher program is responsible for initiating communication to the dispatcher software running on the support equipment, requesting and processing the information required to run the next HTC task, and calling the `execve` system call with the supplied arguments and environment variables required to run the task.

Our design uses a common launcher assigned to all HTC partitions. Since the dispacher may be managing multiple partitions for multiple users, each launcher must send back information that uniquely identifies the partition and node. This allows the dispatcher to respond with an appropriate work assignment. One interesting side effect of the IBM HTC architecture is that because the `execve` system call never returns and the launcher is automatically restarted by the compute node kernel when the task completes, the exit status of the previous HTC task must be reported by the subsequent launcher invocation. Similarly, the standard output and standard error streams remain unchanged through launcher reloads, so the launcher must strategically redirect output in order to produce the individual output files for each task that users expect to receive.

Upon invocation, the launcher first determines its location in the BG/L partition. The location identifier is used to immediately redirect output from the launcher itself to a private log file. Next, the launcher parses its command-line arguments to obtain the dispatcher's connection information and assigned partition callback identifier. A message containing the node's callback information and the return code of the previous task is sent to the dispatcher, which replies with information describing the next task. The launcher then sets the specified working directory, redirects output to the designated files, and executes the program with the requested command line arguments and environment. If the dispatcher does not have a work assignment, the launcher sleeps for a brief period of time and then sends another work request.

On the server side, the dispatcher is responsible for selecting work assignments as requests from nodes are received. Task assignment is performed on a first in, first out basis, respecting vital node characteristics (such as user, nodemode, and accounting project constraints) as well as task walltime requirements. The dispatcher executable is independent from the partition controller, so multiple dispatchers may be started on multiple TCP/IP ports or on multiple servers to provide responsiveness on very large BG/L systems.

V. Task Management System Performance

Executing an HTC task requires a series of potentially time-consuming steps including Cobalt job submission, Cobalt queue wait time, BG/L partition boot time, the distribution of the launcher to the computational nodes, communication between the computational node launcher and the dispatcher, and the transmission of the HTC task executable to the node. The Cobalt job submission, queue wait time, and partition boot time are amortized for all jobs that run on a partition, so for batch runs of many jobs the amount of time required for these initial execution steps decreases in importance. However, because the launcher must communicate with the dispatcher for each HTC task, the scalability of the dispatcher (and the client-server communication scheme in general) is particularly important.

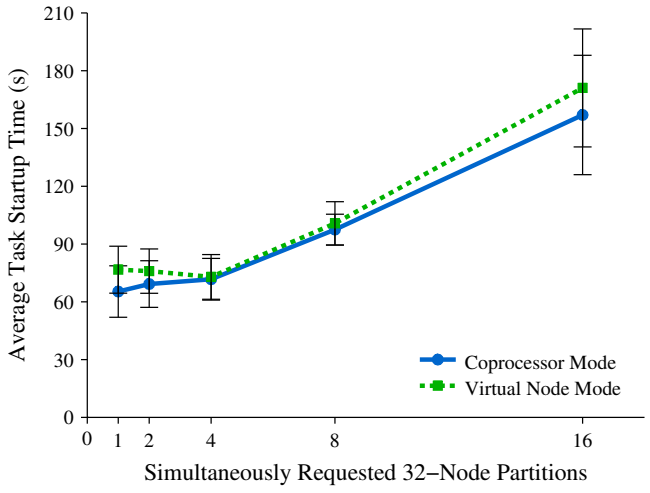

Fig. 2. Average task startup time by number of simultaneously requested 32-node partitions (including Cobalt queue time and BG/L partition boot time) with one dispatcher.

In our implementation, we used a PostgreSQL database to maintain the list of HTC tasks and Cobalt jobs. The HTC GRAM, partition controller, and dispatcher components of the system may all be run on separate machines with shared access to the SQL database. Even with multiple dispatchers, the use of database system locks facilitates atomic job assignment.

A. HTC Task Startup Time

We tested the average amount of time required to start collections of tasks in two configurations: on a cold system where partitions must be booted into HTC mode, and on a system already running idle HTC partitions. To perform each test, we submitted a large number of tasks simultaneously, recorded the submission time and the time the individual HTC tasks were dispatched to the computational node, and subtracted to calculate the startup time for each task. These individual task startup times were averaged and reported based on the number of tasks submitted simultaneously. This essentially describes how long it takes for a task to start running after the user submits the task requests to the queue.

For our first series of tests, we used a single dispatcher to respond to work requests from HTC nodes. The dispatcher used a shared pool of 32 database connections to access the HTC task queue, and all assignments were controlled using a global mutex. The single dispatcher was capable of responding to iterative work requests for up to 512 tasks on an entire BG/L midplane. For larger assignments, we used multiple

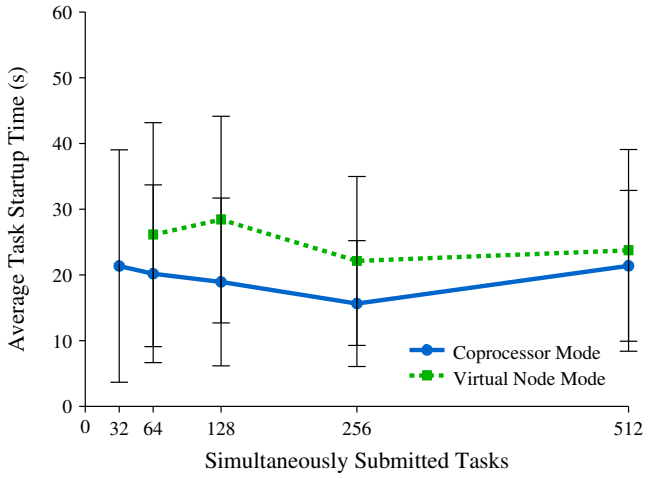

Fig. 3. Average task startup time by number of simultaneously submitted tasks on already running HTC partitions of the same size (with a 30 second launcher polling interval and one dispatcher).

dispatchers on a single machine to verify functionality.

The amount of time required to start tasks on cold partitions is directly dependent on the number of *partitions*, not the number of tasks (see Figure 2). In general, it takes our BG/L system just under one minute to boot a partition and execute a job when directed to do so by the Cobalt scheduler. Moreover, the scheduler does not always process all queued jobs at once. For example, when submitting 16 jobs to Cobalt at the same time, the jobs may be moved from queued to running states in small groups over several dozen seconds. Thus, the launchers on the BG/L partitions are started over a span of time, and that span of time increases with the number of partitions that must be booted.

HTC tasks running in virtual node mode take a slightly longer amount of time to start than tasks running in coprocessor mode. We believe that this may be due to contention at the I/O node while transmitting or booting the node launcher software or socket operations. When observing the work assignment requests, tasks appear to be handed out to nodes that booted at the same time in accumulating sets. The first partitions requesting work will be assigned tasks evenly, but only receive all of the work over a period of time. For example, the first four partitions will show 10 tasks, then the other partitions 10 tasks, and then the first four will increase to 20 tasks, and so on, until all of the nodes on all of the partitions are filled. Thus, using coprocessor mode doubles the number of partitions that request assignments, which decreases the serialization in the task assignment process, and slightly decreases the amount of time required to start tasks.

Because Cobalt queue time and partition boot time only occurs once for each partition, we also tested the amount of time necessary to start jobs on previously allocated partitions. For these tests, we booted HTC partitions and waited until all nodes were posting task requests. We then submitted a large number of jobs simultaneously to place a task on each CPU and recorded the startup time (the time from submission to task dispatch) for each job.

For up to 512 simultaneously submitted tasks, tasks start running between 15 and 30 seconds after their submission to the HTC queue (see Figure 3) with one dispatcher. The polling procedure used by the launcher introduces variance into the job start time. When the dispatcher does not have a task available for the launcher, the launcher stalls for 30 seconds and tries again. Because nodes become free at different times, requests for work from each node in a free partition arrive at different times during a 30-second window. Again, virtual node mode requires more time to start tasks than coprocessor mode.

When dedicating large portions of the system to HTC tasks, the number of socket requests that can be handled by the dispatcher may become a point of contention. To eliminate this potential problem, our system is capable of running multiple dispatchers, and each partition can be directed to contact a specific dispatcher. When using multiple dispatchers, another point of contention appears – the atomic database call that performs the task assignment. When the dispatcher requests a task from the database, it executes a `SELECT ... LIMIT 1 FOR UPDATE` statement that locks the row containing the job record until it is updated with the assignment information. Lock contention at this statement is possible only when many partitions are requesting tasks from a single pool through multiple dispatchers. We use a random offset to select a task from the queue to reduce database lock contention, although occasional lock conflicts do occur and produce a single-iteration stall on the launcher. However, in the expected operational situation where multiple users are running HTC jobs, the database task selection and assignment is already exclusive based on the task properties, so lock contention between users is not possible. Even in the worst-case scenario, with a large number of partitions polling multiple dispatchers for tasks in a single pool, the job start time performance is consistent and similar to that for one dispatcher. For up to 512 nodes on half of a BG/L system, a single dispatcher appears to be sufficient.

B. Task Distribution Overhead

Every time a launcher receives a new HTC task work assignment, the launcher must retrieve the executable from the underlying file system. On BG/L, file system requests from computational nodes are sent over the internal tree network to the partition's I/O node, which proxies the I/O operation to the Ethernet network. Thus, both the tree network and the I/O node are potential bottlenecks when performing I/O operations. Because BG/L does not support dynamically linked libraries, all required libraries must be statically linked to produce the target executable. With many statically linked libraries, the final executable may become quite large. (Our smallest executable is 1.1 MB, a typical executable from our climate modeling codes is 2.2 MB, and more complex applications can exceed 5 MB.) Each executable must be copied from the file system to the computational node independently. Thus, when

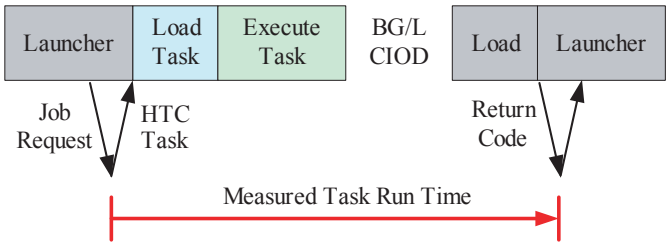

Fig. 4. Task overhead timing methodology.

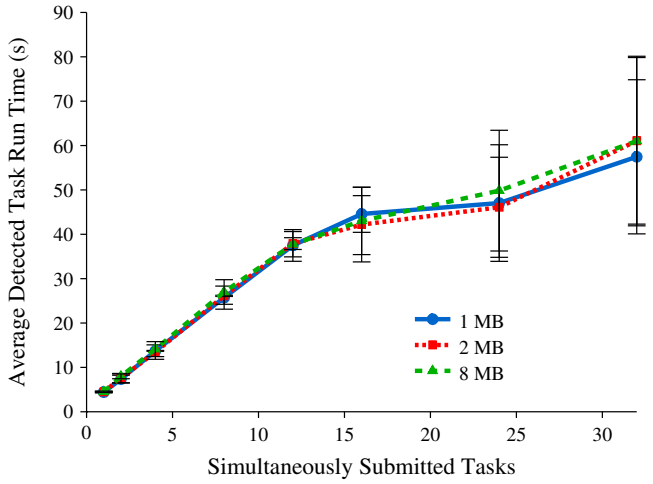

Fig. 5. Average task load time by number of simultaneously submitted tasks on a single partition and executable image size (tasks return immediately, so the reported run time shows overhead).

running large quantities of single-processor HTC tasks, the amount of I/O required to repeatedly launch tasks increases dramatically when compared to standard high-performance computing workloads.

To measure the effect of executable size and task quantity on task load time, we created a series of executables with a variety of image sizes. When run, each executable returns immediately. We ran an increasing amount of these tasks on a single partition (using independent executable files to eliminate I/O node caching effects) and measured the task execution time. This measurement reflects the amount of time elapsed between sending the task assignment response to the launcher and the subsequent launcher return code report (see Figure 4). The reported task run time includes the time required to load the task executable and run the task, as well as the time required by the node's I/O daemon (CIOD) to detect that the executable has finished and reload the launcher, which reports that the previous task has completed.

Our results show that for an increasing number of tasks being placed on a single node, the amount of time required to load, execute, and record the completion of each task increases (see Figure 5). Dispatching a single `return 0` to a single node returns in about 4.42 seconds, but dispatching 32 of them at once increases the return time to 57.46 seconds. The return time of all the tasks increases – the individual run time results do not suggest serial processing of launcher execution on the BG/L system. We believe that this highlights possible contention on the I/O node itself. However, the task turnaround time remained similar for executables of 1 MB, 2 MB, and 8 MB in size, so this contention does not appear to be directly related to the executable size.

When loading large quantities of tasks on multiple partitions, the execution overhead remains consistent. To measure the overhead, we submitted a large quantity of 60-second tasks to the HTC queue with multiple active HTC partitions and calculated the amount of time that elapsed between the task dispatch and the delivery of the return code. In this series of experiments, the system required an average of 46.4 ± 21.2 seconds to to run the task in addition to the task execution time. Although the time to detect the completion of a process is dependent on the receipt of the subsequent task request from the launcher, the measured reload time did not vary with the number of partitions allocated or the number of tasks submitted. Thus, while the load on a single partition may influence task load time on that partition, the load on one partition does not appear to directly interfere with the task load time on others.

VI. Application Results

We evaluated the use of HTC for a Grid-enabled workflow that was previously not compatible with the standard BG/L execution environment. The Daily Meteorological Summary model (Daymet) [11] produces high-resolution meteorological maps of maximum and minimum temperatures, precipitation, and solar radiation intensity for a geographic region, known as tiles, from data collected from observation stations in that region. These maps can then be ingested by other applications, such as the terrestrial carbon cycle modeling application Biome-BGC [12]. Scientists can automate the process using the grid-enabled Grid-BGC [13] interface, which computes maps using Daymet and feeds the results into the carbon cycle model.

Daymet is an embarrassingly parallel application composed of 21 separate executions of 14 sub-applications that aggregate, filter, and interpolate daily meteorologic observations. A run for a geographic area is often decomposed into many independent tiles (see Figure 6). To model the continental United States, approximately 250 tiles are required, and each tile is simulated a year at a time. The time to execute Daymet is bound by the size of the tile and the number of cells that must be computed in the tile. On modern computing equipment, smaller tiles can execute in several minutes while larger tiles can take several hours. Daymet uses multiple processes for parallelism and has traditionally run on Linux clusters, large IBM SMP systems, and other systems that can execute many single processor tasks efficiently.

Before the advent of HTC, employing BG/L to support legacy applications like Daymet would have required re-writing these programs to use MPI. The primary advantages

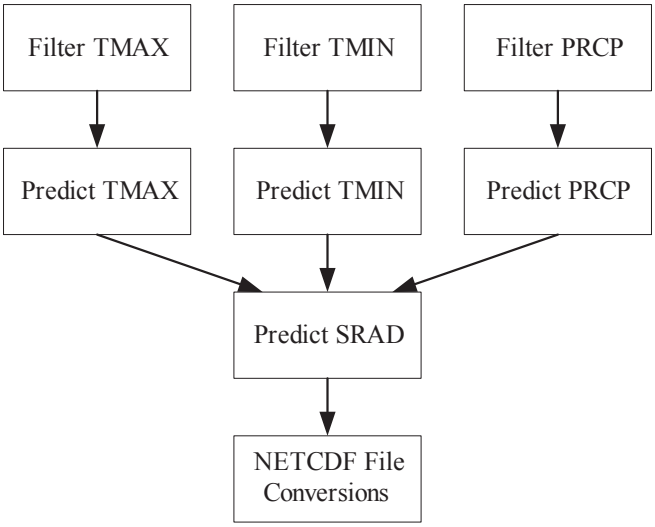

Fig. 6. Daymet single-tile task execution flow (some stages require multiple executables).

to supporting Daymet through HTC, as opposed to writing a specialized MPI wrapper, are simplicity and reusability. As developed by atmospheric scientists, Daymet consists of a series of single-processor applications that are executed as part of a workflow for each tile. It would be possible to write a single MPI wrapper that contains and executes all of the applications as part of a native internal workflow. This would not be particularly difficult, and it would provide additional control of the scheduling and timing of internal tasks with respect to the single application. Several other users on our BG/L system have done this with their applications. However, this approach requires application modification and internal task scheduling, and also limits the usefulness of a running job to the tasks capable of being processed by that program. It is simpler and more efficient to enhance the platform to support running single-processor executables than to suggest that all researchers rewrite their code with MPI wrappers just to be able to execute on the BG/L platform.

We modified Daymet to utilize the HTC-GRAM interface. This interface allows Daymet to submit its sub-tasks to any scheduler that provides a GRAM interface. The modification was straightforward and required the use of globusrun or globusrun-ws to submit the individual tasks in the workflow. We also modified Daymet to utilize Condor-G and DAGMan. Condor-G provides a Condor-based interface to Globus resources managed through GRAM for portability. DAGMan provides a workflow management system that can be represented as directed acyclic graphs. The use of DAGMan can provide an additional level of parallelism in workflows that can execute a set of sub-tasks independently. Support for Condor-G and DAGMan-based workflows can enable the execution of these workflows on BG/L platforms.

When using the Cobalt adapter for GRAM on the NCAR BG/L, it took approximately 20 minutes to execute a small Daymet workflow with 1616 cells for 365 days. Each executable was submitted and allocated on an entire 32-node partition. This allocation strategy consumed approximately 10.67 CPU hours in coprocessor mode and 21.33 CPU hours in virtual node mode to execute a 20 minute simulation. When run in serial, it took 60 minutes using Condor-G and DAGMan due to task scheduling overhead. When optimized to leverage Daymet's three-way parallelism, the run time decreased to 30 minutes, although this used only one processor each on three 32-node partitions. Allocating an entire BG/L partition for a single processor job that uses a fraction of the allocated resources is inefficient. Through the use of HTC, Condor-G, and DAGMan, the workflow was optimized to execute the three tasks in parallel. This optimized single-tile workflow, illustrated in Figure 6, executed in approximately 20 minutes with HTC on a single partition. Resource utilization can be further increased by adapting larger workflows that can utilize nearly all of the allocated HTC resources by running multiple tiles in parallel.

To demonstrate a real-world use of HTC and illustrate how to better utilize the allocated resources, we performed several experiments that focused on scheduling large Grid workflows with HTC. Our first evaluation scheduled 32 simultaneous Daymet workflows of varying tile sizes. Similar to what we noticed during the task management performance analysis, increasing the utilization of the BG/L partitions with multiple workflows increases the turnaround time to complete the workflows. We studied the difference in execution times of a single Daymet workflow in a fully utilized BG/L partition and a partition exclusively allocated for the tile. This workflow was serial and used a single processor. With exclusive access to an entire BG/L partition, the workflow ran in approximately 43 minutes. When scheduled with all 32 workflows, the tile executed in 80.5 minutes. While there is a decrease in resource utilization for a single processor due to the increased turnaround time in workflow execution, the entire BG/L partition is better utilized since all of the available processors are scheduling workflows.

In our second evaluation, we modeled the continental United State's daily meteorological observations during 2004. This large simulation was composed of 263 workflows and 5523 workflow nodes to execute with HTC. We limited the resources consumed by this proof of concept test to a single partition so that the load generated by the Globus client tools did not overwhelm the single job submission node. With HTC and the Globus GRAM adapter, we are able to run the entire Daymet workflow for this large simulation.

VII. Future Work

We intend to merge our high-performance and high-throughput GRAM interfaces into a single submission interface. Our current implementation distinguishes between parallel tasks and HTC tasks, so the user's workflow must submit to either as appropriate. This is not difficult, but it introduces an additional level of complexity that can be eliminated. The modified GRAM interface could run single-processor jobs using HTC mode and parallel jobs using mpirun and the Cobalt

scheduler directly. However, even with a single submission interface, some BG/L-specific submission parameters (such as the coprocessor mode or virtual node mode specification) would still be present.

The partition-level allocation mechanism remains one of the primary limitations to IBM's current HTC architecture. For jobs with any I/O requirements, this effectively constrains all of the tasks on a partition to a single user. While this limitation does not impact demonstration runs or users with obvious bulk computing requirements, it certainly may appear during user operations where workflows occasionally submit single jobs instead of large quantities of jobs. In these cases, an entire 32-node partition must still be allocated for a single task. In the future, if IBM introduces support for `setuid` or `setfsuid`, we intend to modify our task aggregator to pack jobs from multiple users onto shared partitions as appropriate.

We also plan to explore the behavior of our implementation at scale. Our software appears capable of managing our own single-rack BG/L system with HTC workflows running on a full midplane – the maximum portion of the machine that we would administratively relenquish to HTC users. In the future, we would like to more closely examine the scalability of the system and the effect of using multiple dispatcher servers on different machines. In addition, by allowing the launcher to contact a pool of dispatchers (instead of one dynamically assigned dispatcher), we can improve the fault tolerance of the job launch system. In addition, we would like to examine the effect of performing HTC task filtering and assignment during the aggregation process instead of while processing a work request. In the current implementation, requests from booting partitions flood the dispatcher with requests, and the dispatcher responds to each request in isolation. By moving assignment to the aggregation stage, additional scheduling logic could distribute tasks more evenly among available HTC partitions, reducing the contention at the I/O nodes.

Having demonstrated the feasibility for running Daymet on BG/L using HTC, we intend to re-examine our portfolio of other Grid-based climate models for possible deployment on BG/L. The next logical candidate for conversion is the Biome-BGC model, which will allow the entire Grid-BGC terrestrial carbon cycle model to be executed entirely on a BG/L system.

VIII. Conclusions

We demonstrated that by utilizing High Throughput Computing, the power efficiency and processing scale of the BG/L system can be applied to general Grid-enabled scientific workflows, in particular ones that contain large quantities of single processor jobs. Our implementation's integration with the job scheduler provides support for heterogeneous workflows including both HTC and parallel tasks. The performance of the HTC mode, both for the startup latency and the throughput of large quantities of HTC tasks, does not impose significant limitations over commodity computing platforms. Given this new capability, we are planning on exploring the conversion of other Grid-enabled scientific workflows to run on our Blue Gene/L system.

Acknowledgments

We would like to thank Mike Mundy, our IBM Technical Advocate, for his help and advice about the internals of the Blue Gene/L system software.

Computer time and support were provided by NSF MRI Grant #CNS-0421498, NSF MRI Grant #CNS-0420873, NSF MRI Grant #CNS-0420985, NSF sponsorship of the National Center for Atmospheric Research, the University of Colorado, and a grant from the IBM Shared University Research (SUR) program.

References

[1] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas, "Overview of the Blue Gene/L system architecture," *IBM Journal of Research and Development*, vol. 49, no. 2/3, pp. 195–212, March/May 2005. [Online]. Available: http://www.research.ibm.com/journal/rd/492/gara.pdf

[2] "Support for high throughput computing mode," Technical Document 442342092, IBM.

[3] M. Litzkow, M. Livny, and M. Mutka, "Condor - a hunter of idle workstations," *Proceedings of the 8th International Conference of Distributed Computing Systems*, pp. 104–111, 1988.

[4] M. Livny, J. Basney, R. Raman, and T. Tannenbaum, "Mechanisms for high throughput computing," *SPEEDUP Journal*, vol. 11, no. 1, June 1997.

[5] T. Tannenbaum, D. Wright, K. Miller, and M. Livny, "Condor – a distributed job scheduler," in *Beowulf Cluster Computing with Linux*, T. Sterling, Ed. MIT Press, October 2001.

[6] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke, "Condor-G: A computation management agent for multi-institutional grids," *Cluster Computing*, vol. 5, pp. 237–246, 2002.

[7] R. Raman, M. Livny, and M. Solomon, "Matchmaking: Distributed resource management," in *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC7)*, Chicago, IL, July 1998.

[8] A. Peters and T. Budnik, "High Throughput Computing on Blue Gene," presented at Paradyn/Condor Week, Madison, Wisconsin, Apr. 2007.

[9] E. Walker, J. P. Gardner, V. Litvin, and E. L. Turner, "Creating personal adaptive clusters for managing scientific jobs in a distributed computing environment," *CLADE 2006: Challenges in Large Distributed Environments*, pp. 95–104, 2006.

[10] N. Desai. Cobalt Web page. Argonne National Laboratory. [Online]. Available: http://trac.mcs.anl.gov/projects/cobalt

[11] P. Thornton and S. Running, "An improved algorithm for estimating incident daily solar radiation from measurements of temperature, humidity, and precipitation," *Agricultural and Forest Meteorology*, vol. 93, 1999.

[12] P. Thornton, B. Law, H. Gholz, K. Clark, E. Falge, D. Ellsworth, A. Goldstein, R. Monson, D. Hollinger, M. Falk, J. Chen, and J. Sparks, "Modeling and measuring the effects of disturbance history and climate on carbon and water budgets in evergreen needleleaf forests." *Agriculture and Forest Meteorlogy*, vol. 113, pp. 185–222, 2002.

[13] J. Cope, C. Hartsough, P. Thornton, H. Tufo, N. Wilhelmi, and M. Woitaszek, "Grid-BGC: A grid-enabled terrestrial carbon cycle modeling system," *Proceedings of the 11th European Conference on Parallel Processing (Euro-Par)*, August 2005.