A Synthetic Task Model for HPC-Grade Optical Network Performance Evaluation

Sébastien Rumley, Lisa Pinals, Keren Bergman
Department of Electrical Engineering
Columbia University, New York, USA
sr3061@columbia.edu

Gilbert Hendry
Sandia National Laboratories
Livermore, CA, USA

ABSTRACT
With vastly increasing system parallelism, energy efficient data movement has emerged as one of the key challenges in High Performance Computing (HPC). Optics offers the potential for creating system-wide interconnection networks with extremely high bandwidth and energy efficiency. To reap the significant benefits of optical data movement, the interconnect design must go beyond a simple wire replacement to include a fully networked architecture. Simulation is an essential tool in the associated architecture design space exploration. However, simulation requires appropriate models for capturing the relevant interactions between parallel application communication and network operations, as well as the photonic physical layer integrity. An important goal is to keep the simulation as simple as possible to enable a wide design space exploration. In this paper, we propose a traffic generation model that captures the interactions and dependencies between computation and communication for a given application. Our proposed model remains tractable enough to be implemented on top of detailed photonic network simulators as well as general enough to analyze different components of the network under investigation.

Categories and Subject Descriptors
C.4 [**Performance of systems**]: Modeling techniques. I.6.5 [**Simulation and Modeling**]: Model Development.

General Terms
Measurement, Performance.

Keywords
Performance evaluation, high-performance computing, optical interconnects.

1. INTRODUCTION
In recent years, parallelization has emerged as the main enabler of High Performance Computing (HPC) speedup [5]. However, for many applications, the rising levels of parallelization induce a drastic increase in the amount of data movement. For this reason, interconnects linking multiple computing elements have become critical components of the overall HPC system. Many novel architectures have been proposed [1] [20] in response to this problem. Typical modern HPC interconnects span tens of meters and must sustain Terabytes per second of aggregated bandwidth. Given these requirements, photonics is a promising solution due to its wide offered bandwidth and limited signal attenuation over long distances. Although only a small portion of the links are optical in current supercomputers [13], this proportion is expected to increase in the upcoming years as technological advances are leading to cheaper and more efficient optical links.

There is also a great incentive to perform switching operations optically to reduce the need for signal conversions between the optical and electronic domains. If optical switching is performed on Wavelength Division Multiplexed (WDM) data, a common technique for optical transmission, the switching of hundreds of Gb/s can be performed in one single operation. The integration of optical switching in super-computer type interconnects using Micro Electro Mechanical Systems (MEMS) has previously been shown to be beneficial to performance [3] [21]. Emerging technologies, such as Silicon Photonics, have also been proposed to perform optical switching [11] [19].

Optics provides wide bandwidth, quasi-distance independence, and fast, low power switching. Furthermore optics can be exploited to perform unconventional networking operations, such as multicast communications [25]. However, no practical optical memories or buffers have been demonstrated to date, presenting a significant challenge for network design. Optical networks are often presented as a secondary circuit-switched network for increasing bandwidth, or in a segmented form with electronic buffers at the edge [19]. These obstacles complicate the design process of hybrid electrical/optical interconnects. To make the design process even more difficult to define, the design objectives vary with the HPC system target application(s). Some applications primarily require wide bandwidth, while others require fast message passing mechanisms. This necessitates an iterative co-design process using a variety of devices in different configurations that typically lead to estimated performances under a suite of target applications [7].

In order to speedup this iterative process, more interactions between application developers and networking engineers is necessary. Generally, network architects express the capacities of their architectures in terms of bisectional bandwidth and/or latency while application developers tend to describe an application as bandwidth sensitive or latency sensitive. These generic definitions unfortunately provide limited information. Very often two architectures providing the same theoretical bandwidth can result in different application performances. This is especially true in an optical networking context, where network arbitration protocol can play a significant role in the ultimate performance. Recent studies have shown that parallel computer architectures can be affected by long tail latency distributions [9].

IA^3 '13, November 17 - 21 2013, Denver, CO, USA

http://dx.doi.org/10.1145/2535753.2535759

However, such statistics are difficult to obtain, as the latency is application dependent.

In this paper, we propose a new abstract model to address these issues. We start by representing applications as a set of interdependent computation and communication operations. Some computations can take place only after the reception of input data, and resulting communications arise only after the execution phase completes. This approach is similar to the skeleton-based model used by Jannsen et al [15]. However, while skeletons can explicitly represent an application code, they can be labor-intensive or unrealizable, especially for data-driven applications that exhibit irregular behavior. Unlike skeletons, which are derived from real applications, we create random operation sets. By changing these random sets, we can change the type of traffic, the length of the computations, and their interdependencies. We can therefore mimic a broad range of different classes of applications. In particular, by creating random operation sequences based on random tree structures, we intend to capture the various interdependencies between computations and communications present in irregular applications. The main goal of the proposed scheme is to capture the particular structure and dependencies of application traffic without involving the application itself. Given that the simulator is simplified at the traffic level, this releases a margin in terms of execution time and software complexity. This margin can in turn be exploited to consider more hardware specific aspects in the network simulation, in our case, optical networking technicalities. However, it is worth remarking that the approach proposed here is not optics-specific and can be used beyond optical network performance evaluation.

The remainder of this paper is organized as follows. In the next section we compare our approach with existing methodologies. In Section III, we present a model for the task generator of a HPC system and show how it can be parameterized. We apply generated operation sequences to several interconnection architectures in Section IV. In Section V we discuss how our approach can be extended. Conclusions are drawn in Section VI.

2. RELATED WORK

The design methodology of a complex HPC interconnect typically consists of iteratively 1) sketching a preliminary design based on the estimated application requirements; 2) optimizing the interconnect parameters while keeping the design physically achievable; 3) evaluating the performance with modeling, simulation, and/or prototypes [12]. Analytical modeling is of limited use due to dynamic behavior such as congestion-aware or adaptive routing. Simulation is therefore generally considered the main evaluation technique, along with test-beds and prototypes later in the design cycle.

Candidate networks can be simulated with simple traffic models, such as Poisson, which assume no correlation between message arrivals. This gives good insight as to the intrinsic network performance in a dynamic environment (e.g. bandwidth, minimal latency). However, because most application traffic significantly diverges from Poisson traffic, it is difficult to obtain information as to how well the candidate network fits with an application. More complex packet or flow based generation methods have been proposed, in particular to capture the self-similar nature of the traffic in many networks [18]. Soteriou et al. proposed this type of traffic model for networks-on-chip [22]. Another packet generation technique consists of reading trace files of previously executed applications [23]. These complex traffic models, or trace based packet generators, produce flows with similar characteristics to those induced by applications. However, the traffic is replicated a posteriori - this method does not allow analysis of the impact of a message's delay on the application state or on future packet emissions.

To capture these dependencies, a network can be modeled within a cycle accurate simulation platform, such as gem5 [4][24]. On these platforms, HPC sub-systems are described at the instruction set level. Code execution, and therefore all application/network interdependences can be simulated. Because the interconnect is stressed in a realistic way, this cycle accurate approach provides good system scale performance estimates. However, the network's effect on performance might be difficult to isolate, as many elements of the system are modeled simultaneously. Furthermore, such extensive simulations are extremely complex and time consuming to construct and execute, and nearly impossible for extreme scale. For this reason, cycle accurate simulators tend to model the interconnect with a low level of detail (e.g. accounting only for a given delay when a message has to be transported, disregarding the network state).

To capture application/network interdependence while keeping the simulation more tractable, transformed computer code is executed along a coarse-grained network simulator [15]. The portion of code that is purely computation is replaced by a jump in time or a reduced model of the computation, while the network simulator handles communication calls. In this way, the communication patterns can be simulated without significant overhead, while remaining consistent with the original application.

Another simulation approach is to execute application derived benchmarks on real hardware in order to understand the requirements in terms of I/O [6] [10]. This method provides exact measurements in number of bits and execution time, both extremely useful statistics. Similar to our proposed synthetic model, some benchmarks can be configured to stress different aspects of the interconnect. However, this approach is unfortunately only available to HPC systems users. Additionally, as the hardware is "concrete" - different protocols cannot be simulated in rapid succession (the entire system must be reconfigured). Finally, in situ benchmarks are only rarely used to monitor the activity of individual interconnect components, as probes are difficult to insert directly into the HPC system.

These performance evaluation approaches are generally complementary due to the different levels of software infrastructure they require. Packet generation models can be implemented in very lightweight software blocks, and therefore quickly adapted on top of any network simulator. This flexibility places them first in line to test novel interconnect concepts. Application simulators require a greater time investment: new interconnect architectures must be developed and integrated, simulations conducted, and relevant results extracted. These application simulators will typically be used as a second-pass performance analysis. The output of a given performance evaluation approach can also be used as input for another approach. For example, basic results in terms of bandwidth and latency can first be obtained through packet simulation. These bandwidth and latency values are then used as characteristics of the simplified interconnect employed in cycle accurate simulations. The opposite direction is also plausible - cycle

accurate simulations can be used to obtain traces. Fig. 1 summarizes these approaches and their potential interactions.

This description leaves a gap between lightweight independent packet based generators and accurate communication pattern simulators that capture the packet interdependencies but require more software development. A mechanism that both is easy to implement on top of network simulators and accurately generates correlated packets is desirable. For our purposes, this requirement exists in the context of optical networking.

A related system was realized with the PhoenixSim simulator. Graphs representing computation and communication events and dependencies are read from a file and packets are generated accordingly [14]. Here we extend this concept and propose a random generation of inter-dependent events.

Pseudo-random synthetic workloads have also been proposed in other research areas other than network simulation, in particular in the context of task parallel programming. Olivier et al. [16] benchmark the execution of random tree-based workloads using two parallel programming languages (and associated compilers). The performance analysis of parallel programs expressed in terms of directed acyclic graphs (DAG), as reported in [17], shares many similarities with the work we present here. Instead of considering a fixed workload and scheduler as presented in this work, Olivier et al. [17] analyze the impact of compiler and task runtime model on the performance (where the task runtime model is the counterpart to the scheduler in our work). While our final goal is to optimize the hardware, this is not considered in [16] and [17] because their goal is to optimize the scheduling and compilation for a given architecture. The other major difference is that shared-memory multicore architectures are considered in these studies, while we focus on pure message-passing ones.

3. MODEL DESCRIPTION

3.1 General Description

We consider an HPC system with N interconnected nodes, which receives jobs at random points in time. We assume a node to be an independent CPU. This system is composed of N-1 computing nodes, and one *scheduler* node. When a job is received, the scheduler assigns it to one node of the system (let this be called node 1). If no node is idle, the task is queued at the scheduler.

Upon job reception, node 1 executes the assigned *root-task*, starting with the *initialization* phase. Each task may or may not contain independent subtasks. Once the *initialization* phase terminates, node 1 is aware of the number (and type) of subtasks to execute. If there is more than one subtask, node 1 contacts the *scheduler* and asks if idle nodes can be assigned to these subtasks to speedup the task completion time. If such idle nodes exist, the *scheduler* returns a list of idle nodes. Node 1 then contacts each node on this list and delegates a subtask, i.e. sends the data required to start the execution of each subtask. A subtask can in turn be composed of more subtasks. The same scheme repeats recursively, eventually reaching *leaf* tasks, which, by definition, have no subtasks. Leaf tasks also have an *initialization* phase, but this is instead directly followed by an *aggregation* phase. Once this aggregation phase terminates, the aggregated results are transmitted back to the delegating node, or are simply kept locally if the subtask is executed locally.

A task comprised of subtasks similarly has an *aggregation* phase that can be executed only when all results of the subtasks are available. If one or more subtasks have been delegated, the parent task might have to wait for the results. As soon as all of these results are obtained, this *aggregation* phase starts. Upon completion, results are transmitted to the delegating node if the completed task is a subtask. If the task is instead the *root task*, we assume that no transmission takes place. In a more sophisticated version of the model, one node of the system could be designated as the I/O manager, to which final task results would be directed.

Each randomly arising *job* has a tree structure, with a root-task and subtasks as vertices. The dependencies between the tasks form the tree branches. Downward facing edges (toward the leaves) indicate that subtasks cannot start before the parent task's initialization phase ends; upward facing edges (toward the root) imply that the task's aggregation phase depends on the completion of the subtasks. Fig. 2 provides graphical examples of such trees. In particular, Fig. 2b represents a MapReduce type operation, extensively used in data-centers [25]. The tree task representation detailed here is not directly a dependency graph as evoked in [14]. However, such a dependency graph G' can be obtained by applying the following graph transformation to a job tree G. a) Each vertex v in G is divided into two vertices v_i and v_a, representing the initialization and aggregation phases respectively. b) v_i and v_a inherit v's outgoing and incoming edges respectively. c) If v is a leaf, an additional edge $v_i \rightarrow v_a$ is added as the initialization phase precedes the aggregation phase. An example of such a transformation is depicted in Fig. 3. Note that G' is also a directed acyclic graph (DAG) *representing an unfolding sequence of interrelated tasks* [17].

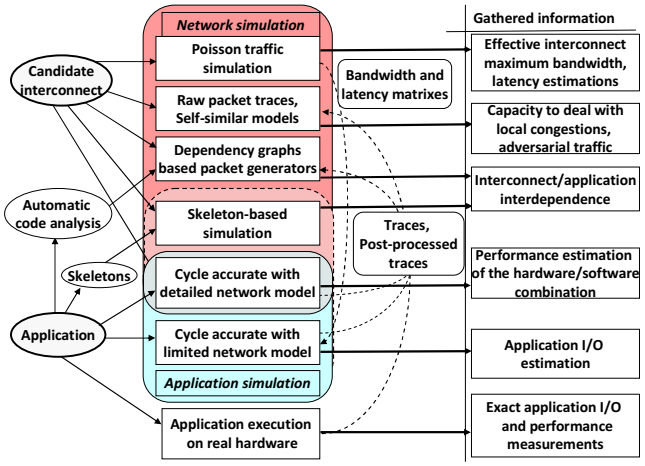

Figure 1. Review of interconnect traffic simulation approaches

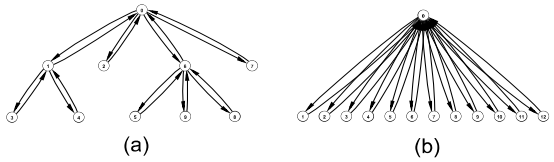

Figure 2. Two examples of tree-represented tasks

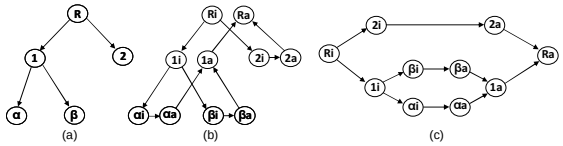

Figure 3. Tree task graph to dependency graph transformation

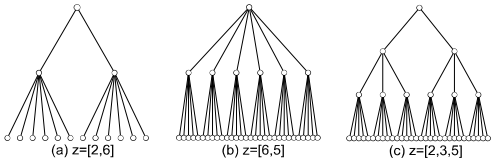

Figure 4. Trees provided by the deterministic tree generator

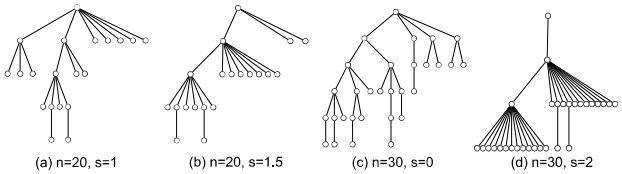

Figure 5. Trees generated by the stochastic generator

3.2 Random task and trees generation

To generate the tree structure, we employ either a deterministic method or a stochastic one. The deterministic tree generator takes a vector, $\boldsymbol{z} = [z_1, z_2, \ldots, z_i]$, as a parameter, and constructs the tree such that each vertex of the i^{th} level of the tree has z_i siblings. Fig. 4 displays several resulting trees for different $\boldsymbol{z}$ vectors. The stochastic method is inspired from the preferential attachment model described by Barabasi et al. [2]. Given a tree graph H, we obtain a new graph H' by adding a new vertex to H and by connecting it to exactly one vertex u already in H. To select u, we compute for each vertex v in H a weight $w(v) = (2deg(v))^s$ where $deg(v)$ designates the degree of v and s is a tree shape parameter. We then perform a weighted random selection. To obtain a tree with n vertices, we create an initial graph with only a single vertex and then apply our $H \rightarrow H'$ transformation $n-1$ times. Several graphs obtained with this method are displayed in Fig. 5. The parameter s is used to change the shape of the graph. When $s >= 1$, graphs with few vertices of higher degree are favored. Conversely, when $s < 1$, vertices with low degree are more common, leading to deeper trees with many levels.

Once the tree is generated, we create as many tasks as vertices in the tree. Formally, a task is described with four values $(\hat{s}_s, \hat{s}_r, \hat{t}_i, \hat{t}_a)$ where $\hat{s}_s$ stands for the number of bits sent when the task is assigned (the data required to start the initialization phase), $\hat{s}_r$ for the size in bits of the task results, $\hat{t}_i$ and $\hat{t}_a$ for the initialization and aggregation phase completion times (in seconds)[1]. Random tasks (i.e. tree vertices) can be obtained by sampling four *random variables* (S_s, S_r, T_i, T_a). Consequently, our two *job generators* take parameters $(\boldsymbol{z}, S_s, S_r, T_i, T_a)$ and $(n, s, S_s, S_r, T_i, T_a)$. Note that to obtain a wider diversity of jobs, (n, s) or $\boldsymbol{z}$ can also be randomly generated. Also note that special sizes are used for the root-task's messages. In the absence of an I/O node, these can be considered control messages.

3.3 Scheduler and delegation model

As previously introduced, the *scheduler* assigns tasks and subtasks. A possible *scheduler* implementation, realized in a similar way in [26], is detailed here. However, other scheduling strategies are possible [17].

[1] As we assume all compute nodes of the system to be homogenous, execution phases can be expressed in time units. In a heterogeneous environment, these would be expressed in terms of operations per second, to be later converted into time depending on each node's computing power.

Our example *scheduler* is the only entry point for arising *jobs*. It maintains a list of flags indicating whether a computing node is allocated or not. It also contains a FIFO for queued jobs. When a task begins, a computing node is randomly selected from those available, the description of the task is sent to this node, and the corresponding busy status is updated on the *scheduler*. If no node is available, the task is inserted into the FIFO. The *scheduler* then reacts to node allocation requests when necessary - as computing node M terminates the initialization phase of its assigned task, it may ask for additional nodes to complete the subtask execution. In our example, M asks for a number of nodes equal to the number of detected subtasks minus one, keeping one task for itself (in this way, node M is kept at least partially busy while expecting results from its delegates). The *scheduler*, upon reception of an allocation request for k nodes, allocates (randomly) k nodes if k nodes are currently available, or, otherwise allocates all remaining idle nodes. The allocated nodes are marked as busy and their IDs sent to M. Reversely, if M detects that an allocated node is no longer needed, it sends its ID back to the *scheduler*. If M was assigned to a root task, it replies to the *scheduler*, too. In both cases, the *scheduler* updates its status list and checks the FIFO. If a task is waiting, it is immediately assigned to the just released node.

Suppose node M is assigned to a task with m subtasks; node M will request $(m-1)$ allocations to the *scheduler*. However, if only $k < (m-1)$ IDs for idle nodes are sent from the scheduler, all subtasks cannot be executed in parallel. Node M therefore sends subtasks *1* through k to each of the k allocated nodes, then execute the $(k+1)^{th}$ subtask itself. Upon completion of this subtask, it assigns the $(k+2)^{th}$ subtask to itself but does not execute it until k answers are received from the k associated nodes. Each time an answer is received, say from node r, an unassigned subtask, if any remain, is sent back to r. If there are no additional subtasks, a release message is sent to the *scheduler* for node r. Once all of the delegates of node M are either released or assigned to a new task, node M starts the execution of the $(k+2)^{th}$ task. This scheme is repeated until all subtasks are either delegated or executed locally.

3.4 Computation and communication footprint estimation

The computation and communication footprints of our model can either be derived for the deterministic case or estimated for the stochastic case. For the sake of notation simplicity, s_s, s_r, t_i and t_a hold for the expectancies of S_s, S_r, T_i, T_a respectively. $\tilde{n}$ is the expected number of vertices in the tree. If the deterministic generator is used,

$$\tilde{n} = 1 + \sum_{k=1}^{length(z)} \prod_{m=1}^{k} z_m$$

The computation time required by a single task is (t_i+t_a). Therefore, the total computation footprint for a compound task is $F_x=\tilde{n}(t_i+t_a)$ (to distinguish between communication and computation footprints, x stands for computation, while c stands for communication). Assuming β job arrivals per second on average, the mean computing load submitted to the HPC is $F_x\beta$. The relative computing load can be expressed as $\rho_x=F_x\beta/(N-1)=\tilde{n}(t_i+t_a)\beta/(N-1)$. ρ_x is the ratio between the offered computation load and the theoretical maximum computational capabilities. It is therefore also an upper bound for the computing nodes' utilization.

The communication footprint is more complicated to estimate. In case of system saturation, fewer nodes are allocated and some subtasks are executed sequentially and locally, which results in fewer communications between nodes. The communication footprint is also determined by the subtask allocation strategy. We consider here the strategy detailed in previous subsection and assume the situation where each delegating node executes one subtask locally. The total number of communication phases between the delegating nodes and the nodes to which tasks are delegated is equal to the number of the leaves in the tree minus one. This fact is true for both stochastically and deterministically generated trees and is easily illustrated with an example. We will examine the case of a deterministic tree (Fig. 4a) with ***z*** vector *[2,6]*. First, the root node keeps one subtask for itself but must communicate the other subtask to an idle node. Therefore to move from level zero to one of the tree, there is one message passed. To move from level one to two, each of the two subtasks spawns six sub-subtasks. However each of the two nodes at level one keep one of their subtasks to execute locally. Therefore, moving from level one to two of the tree results in ten messages. Altogether, there are 11 communications, which is the number of leaves minus one. It should be noted that if a task spawns only one subtask, there is no communication because the parent task will execute this one subtask locally as previously explained. Therefore, each job induces network traffic equivalent to $(L-1)(s_s+s_r)$, where L is the number of leaves in the tree. The number of exchanges with the *scheduler* also depends on the tree structure: only the vertices with degree greater than two induce a delegation process (except the root node that has no parent and delegates if the degree is greater than one). Assuming d delegating vertices, $(L-1) + 2d + 2$ control messages are exchanged. The $(L-1)$ corresponds to the release messages sent after each subtask completes, the $2d$ accounts for the request-IDs and IDs-granted messages exchanged with the mapper; 2 more messages account for the communication with the root node to start and end the job. The total communication footprint for a task, under the maximum delegation assumption, is therefore $F_c=(L-1)(s_s+s_r+\varepsilon) + \varepsilon(2+2d)$, where ε is the size of a control message. The expressions or methods to obtain L and d for each tree generator are provided in the Appendix.

Still considering the case where all potential delegations happen (there are always idle nodes when requested), the average load offered to the interconnect is βF_c. Assuming each node has a maximum access bandwidth of B (bits/s), the relative communication load can be expressed as $\rho_x = \beta F_c/BN$. Similarly to ρ_x, ρ_c provides a relation between the communication load induced by the arriving tasks and the interconnect capabilities. However, as tasks can be kept locally, ρ_c is a bound and not a strict prediction.

3.5 Balanced system bandwidth estimation

In large systems (N large), the ratio between the communication and computation loads $\zeta = \rho_x/\rho_c = [\beta F_x/(N-1)]/[\beta F_c/BN]$ can be estimated as $\zeta \approx B/F$, where $F=F_c/F_x$. In the particular case where $\zeta=1$, i.e. when the interconnect and the computing nodes are equally over- or under- dimensioned, we obtain $B=F$. In other words, F is the bandwidth required to have a bandwidth/computation-balanced system. This definition of a balanced system considers *a utilization point of view only*. In fact, diverse application (or, in our case, generated job) requirements call for different bandwidth/computing resource ratios.

Table 1. Notation summary

HPC system parameters	
N	Number of computing nodes in the system
B	Computing node global I/O bandwidth [bits/s]
ε	Control message size [bits]
Task generator parameters	
s_s	Average task assignment message size [bits]
s_r	Average task termination message size [bits]
t_i	Average task initialization computing time [s]
t_a	Average task aggregation computing time [s]
β	Average task arrival rate [s^{-1}]
$\tilde{n}$	Average number of subtasks in a job
L	Average number of leaves in a job tree
d	Average number of delegating tasks in a job
F_x	Average execution footprint per job
F_c	Average maximal communication footprint per job
F	Job generator footprint ratio - Equilibrated bandwidth reference [bits/s]
Deterministic tree generator parameter	
z	Vector of integers describing the tree span at each layer
Stochastic tree generator parameters	
n	Vertices in the tree
s	Tree shape parameter
Global bounds and metrics	
ρ_c	Maximum relative communication load - Maximal theoretical network utilization - Upper bound for measured network utilization
$\tilde{\rho}_c$	Measured node utilization
ρ_x	Relative computing load - theoretical node utilization - Upper bound for measured node utilization
$\tilde{\rho}_x$	Measured network utilization
ζ	System assymetry

Nevertheless, F is useful in expressing this ratio. If a bandwidth $B = \zeta F$ is present in the system, this means that the interconnect is ζ times more provisioned. As we will see in the next section, varying ζ allows shorter job completion times at the price of network utilization, or vice-versa. Table 1 summarizes the model parameters and notation.

3.6 Speedup bound

The completion time of a job arising in a HPC system depends on the properties of tasks associated with that job and on the state of the system. F_x can be taken as a reference point, as it is the completion time if all tasks and subtasks are executed sequentially. Another reference point is the minimal completion time, which is given by the most costly path from the root vertex in the tree (considering $\hat{t}_i, \hat{t}_a$ as vertex costs). For a leaf task to be executed, the initialization phase of each vertex separating it from the root must be completed. As each of these phases depends on the parent's completion, they cannot be parallelized. Successive aggregation phases leading up to the root cannot be parallelized either. The minimal completion time can therefore be estimated with $D(t_i+t_a)$, where D is the depth of the tree. The maximal speedup compared to sequential execution can also be estimated as $F_x/D(t_i+t_a) = \tilde{n}/D$ (as $F_x = \tilde{n}(t_i+t_a)$).

4. EXAMPLES AND RESULTS

The *scheduler* and tree generators are implemented in the LWSim simulator [12]. In our implementation, the *scheduler* is also responsible for simulating the sporadic arrival of jobs. This is achieved by generating exponentially distributed inter-arrival times with parameter β. In the following reported simulations, 5 sequences of 1500 job arrivals have been simulated, using different pseudo-random number generator seeds.

4.1 Implementation and bounds validation

Simple configurations are first tested to validate our implementation. A simplified system organized around a single switch interconnecting the N computing nodes with bandwidth B is considered. The switch and links latencies are fixed to 200ns and 20ns (so about 4m assuming a light propagation speed of 2e8 m/s). The predictability of the deterministic generator is exploited to validate the model: **z**=[*4,3*] which translates into the following values *ñ=17, L=12, d=5*. Constant values of *20kB* (kBytes) and *40kB* are used for s_s and s_t while $t_l = t_a$=10µs. The size of control messages ε is assumed to be 40bytes. This results in footprints per job of F_x=*340µs* and F_c=*660.92kB*, with *F=15.55Gb/s* as equilibrated bandwidth value (ζ*=1)*. The completion time of a single job running on networks with varying bandwidths is represented in Fig. 6a. It is measured as the time elapsed between the job arrival and the "job complete" message arrival at the *scheduler*. For *N*=2 (sequential execution on a single computing node) this time is 340µs plus the round trip time between the computing node and the scheduler, as expected.

For larger values of N, completion times (and speedups, Fig. 6b) depend on how the ***z=[4,3]*** job is *scheduled* over the *N* nodes. As the job tree has a depth of 3 and all tasks are equal, the maximum speedup is *ñ/D=17/3*. For a given *B*, this speedup is maximized as soon as all possible delegations occur (*N >= L*). The speedup increases to the maximum value as *B* increases (Fig. 6c). As it appears in Fig. 6a, some configurations are clearly disadvantageous and show the limit of the job allocation strategy, especially for low *B*. The case when *N*=4 is a typical example (Fig. 7a). Out of the first 4 subtasks, only three can be parallelized. Therefore, node 1 keeps the fourth task, and executes it after reception of the other three aggregation results. The three siblings of the last subtask are parallelized but because the network is slow, this parallelism leads to critical waiting times. Fig. 7b shows a case where subtask can be distributed more advantageously. Note that the dependency graph structure detailed in Fig. 3c is clearly visible in Fig.7b.

We then tested our implementation with N=100 and different task arrival times. We calculated values for the arrival rate β using $\rho_x = \beta F_x/(N-1)$ such that ρ_x = *10%, 20%,* etc. For ρ_x=*0.1, β=29117*. We use network bandwidth proportional to the balanced bandwidth *F* to obtain $\rho_x=\rho_c$ *(ζ=1)*. Fig. 8 shows for these increasing arrival rates (top axis) and loads (bottom axis) that: a) the node utilization or the computation throughput is increasing almost proportionally until $\rho_x \approx 1$, then saturates. As the bandwidth increases, the node utilization approaches one. b) The deviation between the offered computing load ρ_x (which is also the predicted node utilization) and the effective node utilization $\tilde{\rho}_x$ is limited up to $\rho_x \approx 1$. Beyond this point, the deviation becomes obviously larger as the system saturates. c) The effective network utilization $\tilde{\rho}_c$ first grows linearly but drops much earlier as some computations are achieved sequentially (ρ_c is different for each different bandwidth- the corresponding trends are visible via the dotted lines). d) The effective network utilization deviates early from the bound except when bandwidth is over-provisioned (ζ*=4)*.

4.2 Utilization vs. speedup trade-off analysis

We tested our model with most of the parameters assigned randomly and assuming a larger scale system.

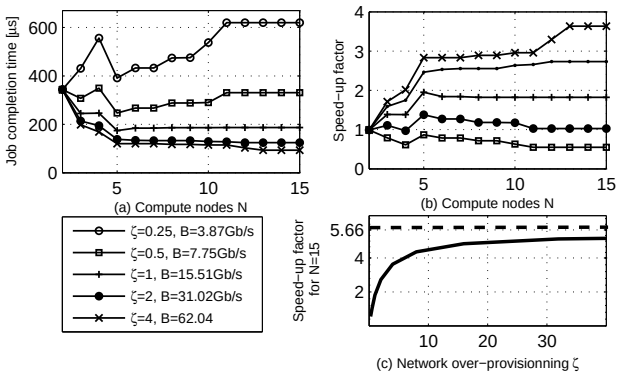

Figure 6. (a) Time required to execute one z=[4,3] task on different architectures for various values of B and N. (b) Corresponding speedup factors. (c) Evolution of the speedup with the bandwidth when N=15.

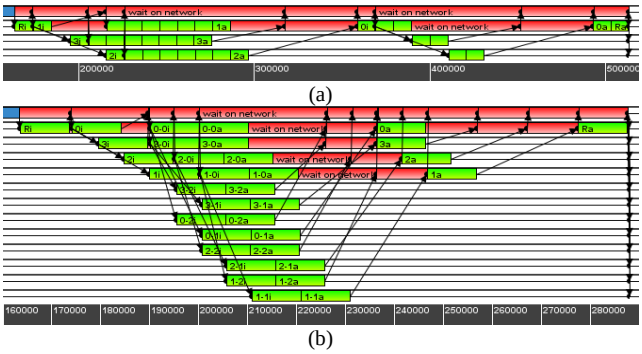

Figure 7. (a) Simulation of one single task (det([4,3]) on N=4 nodes, (b) Same task as (a) with N=13.

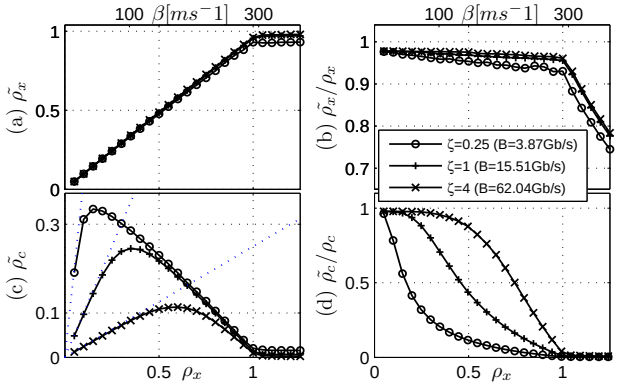

Figure 8. For various task arrival rates β corresponding to different computational loads ρ_x, evolution of: (a) Measured node utilization $\tilde{\rho}_x$ (c) Measured network utilization $\tilde{\rho}_c$. (b) and (d) Comparisons with the theoretical values and bounds.

Task execution times and message sizes are modeled by uniform distributions [*2 µs, 100 µs]* and *[2kB, 128kB]* respectively. Random trees are generated with the stochastic generator. The number of tasks in the tree is in the interval *[25,175]* (so *ñ*=100) and *s* is fixed to one initially. ε is kept to 40bytes. F_x is therefore *10.2ms* and F_c is *9.07MB*. These values give an equilibrated bandwidth *B=F=7.11Gb/s*. We consider architectures with *N*=50 to 1000 computing nodes, and a job arrival rate β of 5000 task per second. We therefore have a computational load spanning between ρ_x=*1.04* (for *N*=50) and ρ_x=*0.051* (*N*=1000).

Figure 9(a) shows the $\tilde{\rho}_x/\rho_x$ deviation. The measured load is lower than expected for a system close to saturation (ρ_x close to 1) or equivalently with a slow network. Fig. 9(b) represents the speedup factor achieved by the system. For *B=10Gb/s* and *100Gb/s*, the speedup factor increases as more computing nodes are present in the system, until a point where idling nodes are always available and all possible delegations can take place. As more bandwidth generally decreases transmission times, node allocation and release messages suffer fewer delays, resulting in a more dynamic node allocation. Fewer nodes are hence required to reach the speedup limit. However, with *B=1Gb/s*, the speedup is always less than one (counter-productive case) and even decreases as more nodes are added to the system. As visible in Fig. 9(c), this is not due to a network overload: the network utilization peaks around 23% for *N=200* and then decreases. It is due to the wide intervals separating the node status update events and the actual start (or end) of the computation phase. The relative time spent in this "synchronization overhead" is depicted in Fig. 9(d). With *B=1Gb/s*, *allocated* nodes idle approximately 70% of the time, leading to poor performances. Note that high overheads are also measured with more bandwidth when the system is facing a more sustained load.

As it appears in the cases reported in Fig. 9, bandwidth (or, more specifically, the bandwidth over-provisioning factor ζ) is the determinant speed-up factor. This fact is confirmed in Fig. 10(a), which represents the speed-up/network utilization trade-off for the previous simulation cases (only one seed per simulation in this case). We can see that using the current scheduler, significant speed-up can only be achieved at the price of very low network utilization. In Figure 10(b) we compare two HPC systems (*N*=300 or *500*) with different bandwidths ranging from *10Gb/s* to *100Gb/s* running two types of random tasks (with *s* set to one or two). The speed-up/network utilization trade-off appears clearly again with this example. The size of the system has almost no effect on the speed-up/network utilization trade-off whereas the random tree shape parameter, *s*, has a significant impact. Therefore to mimic applications with different network utilization profiles, we can simply vary the tree shape parameter.

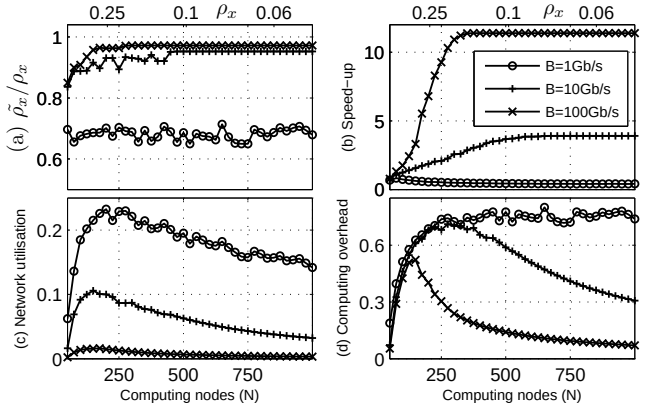

Figure 9. Various measurement with *β=5 task per ms* and random tasks.

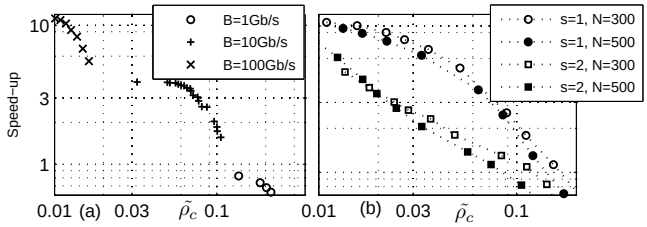

Figure 10. Network-utilization/speedup trade-offs (log. scales)

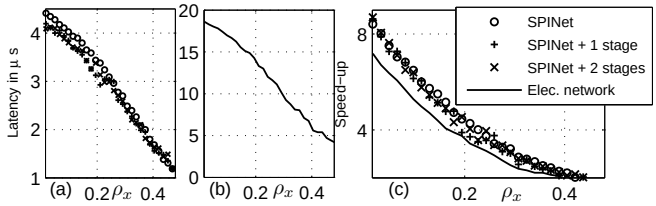

Figure 11. Optical vs. electrical network comparison

4.3 Tree based traffic over optical network

These presented results aim at explaining the detailed characteristics of our random task traffic model. We now achieve our final objective: apply this traffic to different optical networking platforms, analyze how these platforms cope with this new environment, and take appropriate measures if poor performances are reported.

We consider the same random tree task generator configuration, applied to a HPC system comprised of 64 nodes. These nodes are either connected with the single switch network structure (the reference case), or with a multistage optical interconnection network controlled by an in-band signaling mechanism called SPINet [11]. *B* is fixed to *40Gb/s* in the reference case and to *160Gb/s* in the SPINet case. Three different SPINet variants are tested: with the minimal number of interconnection stages (i.e. 6 for 64 ports) or with one or two additional stages. At the network level, the SPINet architecture provides a drastically lower average latency than the reference case (Fig. 11 (a-b)). The additional stages modestly aid to further reduce this latency. However, at the application level, the benefits of the SPINet architecture, while still evident, are not of the same order. Moreover, the modest advantages in terms of latency gained by using multiple stages are not translated into a significant speedup. This demonstrates that optimizing a network considering only latency can be useless, or even counter-productive.

5. EXTENSIONS AND FUTURE WORK

The proposed application model was primarily developed to analyze the impact of unconventional optics-aware novel protocols on interdependent communication phases in a data-center/supercomputer context. In the future, we will leverage our approach to study the impacts of contention resolution schemes or dynamic load balancing on global performance, or more generally to analyze which network topology is best suited for task parallel applications.

Our model has large potential for extensions beyond the applications proposed in this work. An extension could consist of replacing parts of the random tree with a well-known dependency pattern (e.g. an FFT) to capture real application behaviors more closely. Dependency graphs could be generated not out of trees, but as directed acyclic graphs, leveraging an example such as the GGen generator detailed in [8]. The HPC model can also be expanded to consider multi-core nodes or I/O nodes.

As presented in this work, our model is limited by the relative simplicity of the scheduler. Implementing other strategies, typically work-stealing based ones, should allow us to reach higher speed-up and improved computing utilization in our model. This should also provide information about each strategy's communication requirements. Looking further ahead, one can

envision jointly designing the compiler, the scheduler/tasking model and the network.

6. CONCLUSIONS

This paper develops a model for design exploration and performance evaluation of optically interconnected High Performance Computing. The approach uniquely captures the interdependencies between the application and the network mechanisms. It also provides the capacity to tune the network-injected load. Our methodology is both straightforward to implement on top of a network simulator and compatible with optics. Specifically it provides the functionality to include optics at the physical layer while still evaluating the HPC system as a whole. A formal model describing our approach is provided. Numerical results obtained by simulation of various cases exemplify the situations where our approach can provide valuable information. In particular, simulations show that reducing raw latency does not always translate into global performance improvements. Instead, we look beyond latency and analyze traffic dependencies to later make better system-wide decisions to improve overall performance.

7. APPENDIX

The number of leaves L and nodes with degree greater than two d, are straightforward to express when using the deterministic tree generator.

$$L_{\text{det}} = \prod_{k=1}^{length(z)} z_k \qquad d_{\text{det}} = 1 + \sum_{k=0}^{length(z)-2} \begin{cases} z_k \ if \ z_k > 1 \\ 0 \quad else \end{cases}$$

L_{det} (deterministic) is the element-wise product of $\boldsymbol{z}$. For the examples depicted in Fig. 4, L_{det} is 12, 30, and 30 for 4a, b, and c respectively. The number of delegating nodes, d_{det} is the sum of the (n-1) first elements of $\boldsymbol{z}$ that are greater than one, plus one (3,7, and 9 for the examples in Fig. 4). L_{sto} (stochastic) is equivalent to np_1, and $d_{\text{sto}} = p_2 + (n-1)(1-p_1-p_2)$, where p_1 and p_2 denote the probability for a vertex to have degree one or two respectively, and $(1-p_1-p_2)$ is the probability of degree greater than two. A table of values for p_1 and p_2 can be obtained via Monte Carlo simulations.

8. ACKNOWLEDGEMENT

Sandia National Laboratories is a multiprogram laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04- 94AL85000. This research has been partially funded by the National Science Foundation (NSF) Graduate Research Fellowship Program and the IEEE Life Member Graduate Study Fellowship in Electrical Engineering.

9. REFERENCES

[1] Abts, D., Marty, M., Wells, P., Klausler, P., Liu, H, Energy Proportional Datacenter Networks, in *ISCA* (2010).

[2] Barabási, A.-L., Albert, R., Emergence of Scaling in Random Networks. *Science, 286* (5439) (1999).

[3] Barker, K.J., et al., On the Feasibility of Optical Circuit Switching for High Performance Computing Systems, in *SuperComputing* (2005).

[4] Binkert N., et al.. The gem5 simulator. *ACM SIGARCH Compuer. Architecture News* 39 (2) (2011).

[5] Borkar, S., How to stop interconnects from hindering the future of computing. in *IEEE Optical Interconnects Conf.*, (2013).

[6] Borrill, J., et al., Investigation of Leading HPC I/O Performance Using a Scientific-Application Derived Benchmark. in *SuperComputing* (2007).

[7] Chan, C., et al. Software Design Space Exploration for Exascale Combustion Co-design*. in *SuperComputing* (2013).

[8] Cordeiro, D., et al., Random graph generation for scheduling simulations. in *SIMUTools* (2010).

[9] Dean J., Barroso L. A. The tail at Scale. *Communications of the ACM, 56* (2) (2013).

[10] Dinan, J., et al. Dynamic Load Balancing of Unbalanced Computations Using Message Passing. in *IPDPS* (2007).

[11] Dongaonkar, G. et al. Ultra-low Latency Optical Switching for Short Message Sizes in Cluster Scale Systems. in *IEEE Optical Interconnects Conf.*, (2013).

[12] Glick, M. et al. Modeling and Simulation Environment for Photonic Interconnection Networks in High Performance Computing. in *IEEE ICTON* (2013).

[13] Hendry G. The Role of Optical Links in HPC System Interconnects", in *IEEE Optical Interconnects Conf.*, (2013).

[14] Hendry, et al. Circuit-Switched Memory Access in Photonic Interconnection Networks for High-Performance Embedded Computing. in *SuperComputing* (2010).

[15] Janssen, C. L. et al. A simulator for large-scale parallel computer architectures. *International Journal of Distributed Systems and Technologies 1* (2) (2010).

[16] Olivier, S., et al. UTS: An Unbalanced Tree Search Benchmark, in *Workshop on Languages and Compilers for Parallel Computing* (2006).

[17] Olivier, S., et al. Scheduling Task Parallelism on Multi-Socket Multicore Systems. in *Workshop on Runtime and Operating Systems for Supercomputers* (2011).

[18] Paxson, V. and Floyd, S. Wide-area traffic: the failure of Poisson modeling. *IEEE/ACM Trans. on Net. 3* (3) (1995).

[19] Rumley, S. et al. Low Latency, Rack Scale Optical Interconnection Network for Rack Scale Data Center Applications. in *ECOC* (2013).

[20] Scott, S., Abts, D. , Kem, J. and Dally, W.J. The BlackWidow High-Radix Clos Network. in *ISCA* (2006).

[21] Shalf, J., et al. Analyzing Ultra-Scale Application Communication Requirements for a Reconfigurable Hybrid Interconnect. in *SuperComputing* (2005).

[22] Soteriou V., Wang H., and Peh, L.-S. A Statistical Traffic Model for On-Chip Interconnection Networks. in *MASCOTS* (2006).

[23] Tikir, M. M. et al. PSINS: An Open Source Event Tracer and Execution Simulator for MPI Applications. in *Euro-Par* (2009).

[24] Van Laer, A., et al. Full System Simulation of Optically Interconnected Chip Multiprocessors Using gem5. in *OFC* (2013).

[25] Wang, H., et al. Rethinking the Physical Layer of Data Center Networks of the Next Decade: Using Optics to Enable Efficient *-Cast Connectivity. *ACM SIGCOMM Computer Communication Review 43* (3) (2013).

[26] Zydek, D., Chmaj, G., Chiu, S. Modeling computational limitations in H-Phy and Overlay-NoC architectures. *Springer Journal of Supercomputing*, April 2013.