Graph Partitioning for Reconfigurable Topology

Deepak Ajwani
The Centre for Unified Computing,
University College Cork,
Cork, Ireland

Shoukat Ali
Exascale Systems Group,
IBM Research and Development Lab,
Dublin, Ireland

John P. Morrison
The Centre for Unified Computing,
University College Cork,
Cork, Ireland

***Abstract*—Optical circuit switches have recently been proposed as a low-cost, low-power and high-bandwidth alternative to electronic switches for the design of high-performance compute clusters. An added advantage of these switches is that they allow for a reconfiguration of the network topology to suit the requirements of the application.**

To realize the full potential of a high-performance computing system with a reconfigurable interconnect, there is a need to design algorithms for computing a topology that will allow for a high-throughput load distribution, while simultaneously partitioning the computational task graph of the application for the computed topology. In this paper, we propose a new framework that exploits such reconfigurable interconnects to achieve these interdependent goals, i.e., to iteratively co-optimize the network topology configuration, application partitioning and network flow routing to maximize throughput for a given application. We also present a novel way of computing a high-throughput initial topology based on the structural properties of the application to seed our co-optimizing framework.

We show the value of our approach on synthetic graphs that emulate the key characteristics of a class of stream computing applications that require high throughput. Our experiments show that the proposed technique is fast and computes high-quality partitions of such graphs for a broad range of hardware parameters that varies the bottleneck from computation to communication.

***Keywords*-Graph-partitioning algorithms; Reconfigurable topology; Optical circuit switch; co-optimization**

I. Introduction

Optical circuit switches have recently been proposed as a low-cost, low-power and high-bandwidth alternative in the design of high-performance compute clusters (e.g., [19], [10], [3], [17]). At the same time, these switches allow users to configure the network topology to suit the requirements of the application.

The option of configuring the interconnect opens up new possibilities for improvement in topology-aware graph partitioning approaches. Instead of asking the question "given an application graph G, how would you partition it on a set of compute nodes connected in topology H?" we are wondering "given an application graph G, how would you best interconnect the compute nodes to elicit the best possible partitioning of G from your favorite graph partitioner?" This research addresses this question by formulating an iterative strategy for co-optimizing the partitioning of the application graph and the configuration of the network topology.

There are two constraints that further complicate this issue. In a real system, a compute node has only a fixed number of ports to connect to the reconfigurable switch. Secondly, the reconfigurable switch has a limit on the maximum number of simultaneous links that it can maintain. Therefore, as an unavoidable part of our strategy, we also try to minimize the maximum traffic on the interconnect while satisfying the above two constraints. Because our framework attempts to co-optimize topology configuration, application partitioning and interconnect routing, we refer to it as <u>TPR co-optimizing framework</u>.

Our approach is not tied to a particular communication pattern within the application. In our experimental results, we show performance gains for thousands of application graphs randomly selected, (with random communication patterns) from within the class of stream computing applications. Please note that our algorithm works for general graphs, even though our experiments are done on class of graphs that emulate stream computing applications.

The rest of this paper is structured as follows. We present the notations, definitions, the problem addressed and our key contributions in Section II. Section III describes our main framework together with all the details of the individual steps. Our experiments are reported in Section IV and the related work is described in Section V. We conclude with future research directions in Section VI.

II. Preliminaries

In this section, we describe some of the notations and definitions used in this paper and formally describe our problem and our contribution.

A. Notations

We refer to application graph as $\underline{G(V_G, E_G)}$ (or simply $\underline{G}$) and to avoid tedious notation, also use the same notation for contracted application graphs. The notation $\underline{H(V_H, E_H)}$ (or simply $\underline{H}$) is used to refer to the topology graph. The elements of V_G are referred to as <u>vertices</u> while elements in

This research was supported, in part, by an Enterprise Partnership Scheme grant co-funded by IBM and the Irish Research Council for Science, Engineering & Technology (IRCSET).

V_H are referred to as nodes or compute nodes. The notation $\underline{N_P}$ denotes the total number of processors in the supercomputer. Since the nodes in the topology graph correspond to the actual compute nodes in the architecture, we have $N_P = |V_H|$ (although they need not all be connected or have some computation load). We are interested in the mapping of vertices in application graph to nodes in the topology graph. The weight of a vertex or node u is denoted by $\underline{w_v(u)}$, while the weights on an edge e of either the application graph or topology graph is referred as $\underline{w_e(e)}$.

B. Problem Definition

We are given a computational task graph $G(V_G, E_G)$ where the vertices denote computational kernels and the edges capture the dependencies between the different computational kernels.

The weights on vertices denote the average amount of computation that needs to be performed at the corresponding kernel to produce one element of output. Similarly, the weight on an edge represents the average amount of data transfer between the kernels (corresponding to the two incident vertices) to produce one element of output.

We assume that the compute nodes in the high-performance system are identical with the same processing speed (hereafter denoted by $\underline{S_{\text{comp}}}$). These compute nodes are connected through a reconfigurable switch, which can alter the topology to suit the application. We also assume that bandwidth on all links connected through the reconfigurable switch is identical (denoted by $\underline{S_{\text{comm}}}$).

In order to run the application on the system, we need to map each vertex $v \in V_G$ to a compute node and route each edge $e \in E_G$ along some path in the network topology. Let $\underline{\mu(v)}$ be a mapping that specifies the compute node to which a particular vertex v is mapped. Let $\underline{\rho(e)}$ be the sequence of communication links that are used to route an edge e in E_G. Given such a mapping and a routing scheme, the computation load on a compute node $P_i \in V_H$ is

$$\underline{w_v(P_i)} = \sum_{(u \in V_G) \wedge (\mu(u) = P_i)} w_v(u)$$

and the communication load over a link $e \in E_H$ is

$$\underline{w_e(e)} = \sum_{(e' \in E_G) \wedge (e \in \rho(e'))} w_e(e').$$

Since all computation over the nodes and communication over the links happen concurrently, the throughput is constrained by the slowest element. We define the throughput of a node P_i to be $S_{\text{comp}}/w_v(P_i)$ and the throughput of a link e to be $S_{\text{comm}}/w_e(e)$. The compute throughput of the system is the minimum throughput of a node and the communication throughput of the system is the minimum throughput over a link. The throughput generated by the overall system is the smaller of the compute throughput and the communication throughput. Note that our definition of throughput arises out of stream computing applications, where we view the compute nodes and communication links as processing units running concurrently so that the overall throughput is equal to the throughput of the slowest processing unit (similar to the throughput of a fetch-decode-execute pipeline where fetch, decode and execute stages run concurrently). Nonetheless, other problem-specific definitions can be used (with an accompanying change to the performance vector in Section III-B3).

In a real system, a compute node has only a fixed number of ports to connect to the reconfigurable switch. Let this constraint be called the max-degree constraint, denoted as $\underline{\Delta_{\max}}$. Also, the reconfigurable switch has a limit on the maximum number of simultaneous links that it can maintain. We refer to this limit as max-edges constraint, denoted as $\underline{E_{\max}}$. Thus, the switch can configure any topology that satisfies the constraints that maximum degree in the topology is no more than $\Delta_{\max}$ and the total number of links is not more than $E_{\max}$. Note that these constraints on the space of configurable topologies are very natural and can easily arise in many other applications.

Our goal is three-fold.

- Compute a network topology graph H that is likely to elicit a high throughput mapping for the application graph G.
- Compute a mapping of vertices in V_G to nodes in V_H to achieve a high computation throughput.
- Compute a routing scheme for edges in E_G to communication links in H so as to minimize congestion and thereby provide high communication throughput.

A good topology is one that allows a mapping and a routing scheme to yield a high throughput (ideally close to the optimum). Since the definition of a good topology depends on the difficult problems of computing good mapping and routing schemes, it is not easy to compute. We therefore propose a framework where we derive a good initial topology based on the structural properties of the application graph and then iteratively improve this topology by performing local modifications.

Note that although the connections created by optical switch are directed in nature, we treat them as undirected. This is because engineers invariably pair these optical cables to keep the routing protocols simple. Often, the two optical fibers in an optical cable are used for making the data-transfer bidirectional.

C. Key Contributions

Our key contributions are as follows.

1) A new framework that exploits reconfigurable interconnects in order to iteratively co-optimize the network topology, the partitioning and routing schemes to maximize throughput for a given application.

2) A novel way of computing a high-throughput initial topology based on the structural properties of the input graph without explicitly identifying those properties. This topology is referred to as "initial" because it is used as a seed for the framework in item 1 above.
3) A statistically rigorous experimental analysis of the goodness of our technique.

D. Definitions

Given an edge $e = \{u, v\}$, we define the *expansion* of an edge to be

$$\Xi(e) = \frac{w_e(e)}{w_v(u) \cdot w_v(v)} \tag{1}$$

We refer to the denominator in the expansion term as *vertex-product*, i.e., the vertex-product of an edge $e = \{u, v\}$ is $w_v(u) \cdot w_v(v)$.

Contracting an edge $e = \{u, v\}$ means to replace vertices u and v by a new vertex w such that $w_v(w) = w_v(u) + w_v(v)$. All edges of the form $\{u, x\}$ or $\{v, x\}$ for $x \in V$ are replaced by $\{w, x\}$. If both edges $\{u, x\}$ and $\{v, x\}$ exist, $w_e(\{w, x\}) = w_e(\{u, x\}) + w_e(\{v, x\})$ in the contracted graph.

A matching $M \subseteq E$ is a set of edges that do not share any common vertex, i.e., the graph $G(V, M)$ has maximum degree 1. Contracting a matching refers to contracting all edges in the matching.

III. Our Framework

In this section, we propose our framework for co-optimizing the network topology, the partitioning and routing scheme. Co-optimization is needed because these three goals are inter-dependent. In the first phase, our framework computes an initial topology, and corresponding partitioning of the application graph, and an accompanying routing scheme for the computed topology. In the second phase, we perform a number of iterations where each iteration alters the topology and computes both a re-partitioning for the modified topology and an accompanying re-routing. We repeat this iterative procedure till there is no improvement in throughput for a pre-specified number of iterations.

Our framework consists of the following steps.

1) Computing an "initial topology". Here we first compute an architecture-oblivious partitioning of $G(V_G, E_G)$ so as to minimize the maximum volume of data incident to a subdomain (i.e., a partition), together with the traditional objectives of balancing the work load on partitions and reducing the total cut-size. No attempt is made to satisfy the max-degree and max-edges constraints. Then we derive a network topology that accommodates the architecture-oblivious partitioning while satisfying the max-degree constraint on the maximum degree and max-edges constraint on the total number of edges. This topology is referred to as "initial" because it will go through a number of iterative improvements later.
2) Computing an architecture-aware partitioning. We calculate a partitioning of the graph to maximize the throughput on the given topology.
3) Computing a low-congestion routing. We compute a routing scheme to determine how each edge in the graph G can be routed in the given topology so as to minimize the maximum congestion over any link. Note that we do not allow the path to be split as that will require adaptively splitting the data-stream at run-time and a system may not have such capabilities.
4) Performing "TPR co-optimization" In the last phase, we evaluate the existing topology configuration, partitioning and routing, and decide if further optimization is needed. If yes, we repeat the following steps until there is no improvement for a pre-specified number of iterations.
   - Modify the network topology based on the existing partitioning and routing of the application graph.
   - Perform steps 2 and 3 to re-partition, from scratch, V_G and re-route E_G for the modified topology.

Since we co-optimize in the above framework for **t**opology configuration, **p**artitioning and **r**outing, we refer to it as a TPR co-optimization framework in our paper.

Since many of the above mentioned problems are NP-hard in general, we need heuristics to solve them efficiently. We now present our heuristics for engineering a good solution for the TPR co-optimization.

A. Computing Initial Topology

A fundamental question that we address in this section is given an application graph, what topology will yield a high throughput. Our first idea was to identify key structural properties of the application graph, use these properties to classify the input graph into a fixed number of categories, and dedicate a possibly separate scheme for each category. However, the number of such properties needs to be very low for such an approach to be effective and such an approach may not be scalable. We therefore use a condensed graph based approach to determine a good topology.

1) Computing the Architecture-Oblivious Partitioning: Given the computational task graph $G(V_G, E_G)$ of the application, we first partition it into N_P subdomains to achieve the following objectives.

- Balance the *load*, defined as the sum of all vertex weights mapped to a subdomain, across all subdomains. This is to ensure that all compute nodes are adequately utilized, assuming there is enough computational load to require all partitions.
- Minimize the total *cut-size*, i.e. the total weight of all edges between vertices in different partitions. This

ensures that inter-partition communications will be minimized.

- Minimize the maximum subdomain weighted degree, i.e., the total weight of all edges in G that have exactly one incident vertex in the subdomain This objective was chosen in the hope that the weighted degree of a subdomain is correlated to the actual degree of the subdomain.

We use the graph partitioning library METIS [11] ver. 5.0 for this purpose with a random seed, maximum imbalance of 1.05 and the option for minimizing the subdomain degree. The last option relies on the algorithm by Selvakkumaran and Karypis [20]. (While we chose to use METIS for this research, other graph partitioning software could have been used as well, e.g., Chaco [7], JOSTLE [24], Scotch [16] and Zoltan [4]).

Figure 1 illustrates some of the steps involved in creating the initial topology. Please refer to that as needed.

2) Creating Connected Condensed Graphs: Based on the topology-oblivious partitioning, we condense the input graph G. There are as many vertices in the condensed graph as there are nodes in H (i.e., the required number of partitions or subdomains of G). An edge exists between two nodes P_i and P_j if and only if there exists an edge $\{u, v\} \in G$ such that vertex u is mapped to partition P_i and v is mapped to partition P_j. The weight of a node $P_i \in G_c$ is the sum of weights of all vertices in G that are assigned to partition P_i and the weight of an edge $\{P_i, P_j\}$ is the sum of weights of all edges $\{u, v\}$ in G such that u is assigned to P_i and v is assigned to P_j.

In our experiments, we found that the resulting condensed graph does not always span all N_P nodes in G_c. Therefore, we first ensure that G_c is connected and spans the entire graph. To this end, we connect the different components by inserting additional edges. The weight of the newly inserted edges is 0.

3) Satisfying the Constraint on Maximum Node Degree: Next, we impose the constraint that the degree of each node in G_c is less than or equal to $\Delta_{\max}$ and later, we ensure that the total number of edges (i.e., the physical communication links) in the topology is at most $E_{\max}$, ensuring that switch does not have more edges to it than are possible. A major consideration in removing the edges from the graph, to satisfy these constraints, is to avoid creating bottleneck links.

Let us call a node *heavy* if its degree is greater than $\Delta_{\max}$. We call an edge *strongly heavy* if both its incident nodes are heavy and *weakly heavy* if only one of its incident node is heavy.

We meet the max-degree constraint by repeatedly removing edges till all nodes and edges are light, implying that all constraints on degree are satisfied. For any H edge, e_H, (i.e., physical communication link) that we remove, we re-route, along a minimum-congestion path, any G edges that were mapped on e_H.

First, we consider all edges in increasing order of weight and remove those strongly heavy edges that leave G_c connected. When removing an edge $\{P_i, P_j\}$, data-streams that were getting routed along this path need to be re-routed through a minimum congestion path. Since the graph G_c is still connected after removing this edge, the existence of a path between P_i and P_j is guaranteed.

Thereafter, we consider all edges in the increasing order of weight and remove those weakly heavy edges that leave G_c connected. As before, the flow along these edges is re-routed through a minimum congestion path.

If there are still some heavy vertices left, we remove an arbitrary weakly heavy edge and let the graph be disconnected into two components. We pick the minimum weight edges $\{u, v\}$ and $\{w, x\}$ in the two components, remove them and insert $\{u, w\}$ and $\{v, x\}$. This transformation connects the two components, preserves the degree of the vertices and does not increase the total number of edges. As before, the new edges are initialized with a weight 0 and the flow along the removed edges is re-routed through the minimum-congestion path.

4) Satisfying the Constraint on Maximum Number of Edges: Our approach here is similar to that for satisfying the constraint on maximum node degree. First, we consider all edges in increasing order of their weight and remove them if they leave the graph connected. If this still does not satisfy the constraint on total number of edges, we let the graph be disconnected and re-connect it using edge swaps as before.

The goal here has been to preserve as much structural information from the condensed graph as possible and re-route as little traffic as possible.

B. Partitioning Input Graph for the Given Topology

Our scheme for partitioning the input graph for a given topology is based on a multilevel scheme. Multilevel techniques (e.g., [11], [24], [7], [16]) have been a big success both from the scalability point-of-view as well as for providing high-quality partitions.

Our experiments with other approaches for computing the partitioning for a topology suggested that local search heuristics could significantly improve the quality of the partitioning (as measured by the resultant throughput), but for the local search heuristics to be effective, they need to be able to work at coarser levels of graph as well. At the coarser levels, these localized heuristics can move bigger chunks of graphs around.

A multilevel scheme for graph partitioning consists of three phases. In one round of the *coarsening* (or contracting) phase, we identify matchings $M \subseteq E_G$ and contract the edges in M. These rounds are repeated till the number of vertices is smaller than some pre-defined threshold. It is followed by an *initial partitioning* phase where some expensive techniques can be used to partition the graph into required number of subdomains. In the *refinement* phase, the

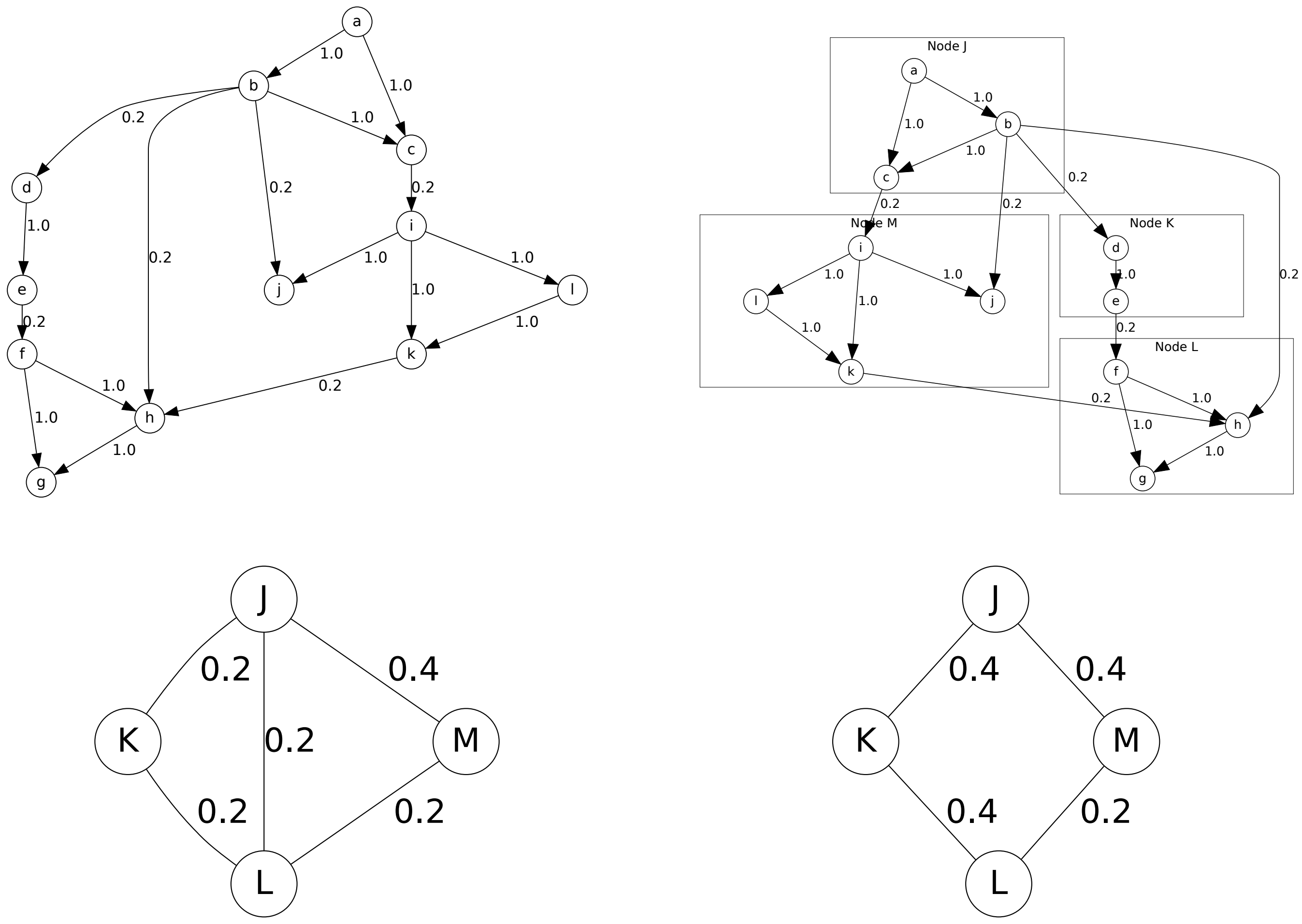

Figure 1. This figure shows various steps in computing an initial topology of the application graph shown in top left for an architecture that has 4 compute nodes, a max-degree constraint of at most two physical communication links per compute node, and a max-edges constraint of at most 4 physical communication links. The top right shows an architecture-oblivious (i.e., degree and edge constraints ignored) 4-way partition. A condensed graph of the architecture-oblivious partitioning is shown in bottom left. The initial computed topology is shown in bottom right, where both max-degree and max-edges constraints are met, and the maximum congestion on all 4 physical communication links is optimized.

matchings are uncontracted. After uncontracting a matching, the refinement algorithm uses some local search heuristics to improve the partitioning objective.

The multilevel approaches work well because the coarsening phase preserves the structure of the input graph while reducing its size. Therefore, a good partitioning at a coarser level leads to a good partition at the finer level as well. Please refer to extensive literature on multilevel partitioning (e.g., [11], [24], [7], [16]) for more details.

In our case, the multilevel approach has the following components: (a) coarsen the graph, (b) compute an initial mapping of vertices to topology nodes and compute a routing scheme for the coarsest graph (c) refine where the local search heuristics in the refinement phase can potentially move the vertices between different partitions as well as reroute the traffic through a different path. We give more details of these components below.

Note that in this section, we treat the topology graph as unweighted, i.e., each node and edge in the topology graph has a weight of one. This is to ensure that the current partitioning is not influenced by partitioning done in previous iterations and is re-computed from scratch.

1) Coarsening: For the coarsening phase, matchings based on edge expansion have been shown to be more effective than those based on edge weights [8]. We therefore use a greedy maximal matching based on edge expansion for contracting the graph.

We start with an empty matching $M = \emptyset$. We consider the edges in increasing order of their expansion values. If both end-points of an edge e have degree 0 in $G(V, M)$, we insert the edge e in M. We coarsen the graph till we have only $\max\{P^{1.5}, 100\}$ vertices left, where P is the desired number of partitions.

2) Initial Partitioning: The initial mapping of the coarsened graph to the topology graph is computed by recursively bisecting *both* of these graphs to minimize the total edge-cut and then mapping the bigger part of the coarsened graph to the bigger part of the topology and the smaller part to smaller. A part is considered bigger if it has a higher load (total weight of all vertices in it) or if the two loads are equal, than it has more edges. The intuition behind this approach is that it maps the sparse cut in the coarsened graph to the sparse cut in the topology in order to avoid (or alleviate as the case may be) communication bottlenecks. The definition of bigger graph is particularly relevant for the topology graph as that is treated as unweighted and is typically small. Thus, the two partitions have the same number of nodes, but the partition with higher number of edges can deal with higher load partition of the coarsened graph as it can balance the load among its nodes better. Note that this approach is similar to the one used in Scotch [16].

We use METIS to partition the two graphs. However, we observed that for very small graphs (less than 8 vertices), METIS tends to put all nodes in the same partition, irrespective of the load balance required. To avoid this, we do a brute-force partitioning for small graphs. We consider all cuts that create balanced partitions and select the one with minimum edge-cut.

For routing the traffic, we first route using the shortest path metric and then update the routes using minimum-congestion path. For the shortest path routing, we consider all edges in the coarsened graph. If the incident vertices of an edge are mapped to different nodes in the topology graph, we route the data-stream between the vertices through the shortest path between the mapped processors in the topology. To update it using minimum-congestion routing scheme, we consider the edges of the coarsened graph in decreasing order of their weight. If routing the flow of the edge through the minimum congestion path results in increased throughput, we change the route of the corresponding data-stream. Considering the edges in decreasing order of weight allows for the lower weight flows (data-streams with fewer expected data) to route through less loaded links after the distribution of heavy weight flows broadly defines the load on the links.

The intuition behind using the shortest path routing first is that the shortest path routing minimizes the total flow (summed over all links) and then the min-congestion routing performs relatively minor modifications to distribute this flow more evenly at the cost of increasing the total flow by a small amount.

3) Refinement: The coarsened graph is projected back to the original by assigning vertices u and v that were merged to produce vertex w, to the processor to which w was assigned at the coarser level. Similarly, the flow for edge $\{u, x\}$ or $\{v, x\}$ for $x \in V_G$ is routed through the same path as $\{w, x\}$ in the coarser graph. This is then followed by a greedy local update procedure which either re-maps a vertex to another processor or re-routes a flow to improve the performance of the system till it converges to a local optimum.

For this local update step, we consider a performance vector. For a topology graph $H(V_H, E_H)$, this vector has $|V_H| + |E_H|$ entries, corresponding to its vertices and edges. For each node $v \in V_H$, we insert $S_{\text{comp}}/w_v(v)$ in the vector to denote the throughput obtained if v was the bottleneck vertex. Similarly, we insert $S_{\text{comm}}/w_e(e)$ for each edge $e \in E_H$ to denote the throughput obtained if e was the bottleneck. A performance vector is obtained by sorting the values in increasing order. Note that the overall throughput is the first entry in this vector. We say a performance vector $\mathcal{P}'$ is better than a performance vector $\mathcal{P}$ if $P'[i] > P[i]$ and $P'[j] = P[j]$ for $0 \leq j < i$. Clearly, the best performance vector also yields the best throughput.

Optimizing the whole performance vector rather than just the throughput (which is the first entry in the performance vector) helps in a more even distribution of load and communication. In our experiments, we found that quite often, it also improves the throughput. This is because alleviating the load on the neighbors of the bottleneck processor or link allows some load to move to them, thereby improving the throughput.

Our greedy local update procedure considers a series of steps involving vertex re-mappings and re-routing of flows and selects the update that results in the best performance vector among all steps considered. The best step is actually executed and the partitioning information and/or routing tables are modified. The procedure of greedily selecting a locally best update and executing it is repeated till the process converges to a locally optimum performance vector.

The steps considered for the local update procedure are as follows:

- For each vertex $v \in V_G$ mapped to processor P_i, we consider re-mapping it to all neighboring processors of P_i in the topology graph H.
- For each edge $e \in E_G$, we consider re-routing it through a minimum-congestion path.

C. Reconfiguring the Topology to Improve Partitioning

In this section, we show how we modify our topology by identifying the key bottleneck with the partitioning computed in the previous iteration. Note that these changes do not necessarily make the topology better and can be undone in the next iteration.

Let the current iteration be i and the current topology graph be H_i. We first identify whether computation or communication is the bottleneck. If computation is the bottleneck, it could be because the partitioning algorithm could not move the vertices of the input graph to lower weight nodes (partitions) for the fear of increasing cut. (We understand that the reason might very well be the fact that the partitioning algorithm is just not good enough. However, we are choosing to make this assumption now and later test it in our experiments.) This implies that in the topology, we need more edges between low-weight nodes and high-weight nodes so as to alleviate the high-cut concerns of the partitioning algorithm and thereby enable it to find a more balanced computational load distribution in the *next* iteration. We therefore select an edge between two low-weight nodes and one between two high-weight nodes and perform a swap operation. Note that this swap can only be done in a setting where topology is reconfigurable. Furthermore, this swap, and the accompanying considerations, are among the features that make this framework a co-optimization because identification of issues with the partitioning lead to local fixes in the topology, which elicits another round of partitioning. A swap operation between edges $\{u, v\}$ and $\{w, x\}$ involves removing the two edges and either inserting $\{u, w\}$ and $\{v, x\}$ or inserting $\{u, x\}$ and $\{v, w\}$.

The edge e_{high} with the lowest expansion value ($e_{high} = \min_{e \in H_i} \Xi(e)$) satisfies the requirements of a low-weight edge between high-weight nodes. To select the edge between two low-weight nodes, we consider the edges in increasing order of their vertex-product and identify the first edge e_{low} that can be swapped. An edge pair can be swapped if all of the following conditions are true.

- The two edges are vertex disjoint.
- Swapping the edges leaves the graph connected.
- The new edges to be inserted in the graph in the swap operation are not already present.

We then swap the edges e_{high} and e_{low} in H_i to obtain the topology H_{i+1} for the next iteration $i + 1$.

If on the other hand, the bottleneck is the communication over a link e_b in the topology, we try to guess the bottleneck cut and introduce more edges between the two sides of the cut to alleviate the bottleneck.

The bottleneck cut is identified by sorting the edges in decreasing order of their weight and then removing the edges till the graph gets disconnected. After the two sides of the cut are identified, the edges are re-inserted into the graph. We then find low-weight edges from both sides such that they can be swapped and swap it to obtain H_{i+1}. If one of the components has no edges (e.g., it might have only one node), we identify the lightest weight edge-pair that can be swapped and swap it to get H_{i+1}. In our experiments, we observed that the last case happens quite often as communication bottleneck is caused by a node that needs to send vast amount of data to the remaining nodes in the topology and it does so by distributing the communication load roughly equally along all incident edges.

A major problem with the above technique for modifying topology is that it tends to get stuck in small cycles, i.e., $H_{i+c} = H_i$ for a small c. This restricts the number of different topologies explored quite significantly. In order to avoid this problem, we introduce a measure of randomization. Even if an edge pair can be swapped, we ignore it with a certain probability and keep searching for a new pair.

IV. Experiments and Results

A. Summary of Results

To the best of our knowledge, we have no direct competitors of our TPR co-optimizing framework because we are not aware of any algorithms that partition graphs for a reconfigurable topology platform while co-optimizing cut, load balance and flow at the same time. This made it difficult to compare against other efforts. Also, since most variants of graph partitioning are NP-hard, it is very difficult to ascertain the quality of our solutions by comparing against a known optimum for interestingly large problem sizes.

We therefore compare our approach with the extreme case of computation load being equally balanced on network with no communication restrictions (i.e., cut minimization does not matter in this extreme case because it is assumed that there are links of infinite bandwidth between all compute node pairs). In our experiments, we found that in the settings when our approach is computation bound, our throughput results on our simulated streaming applications are within a small constant factor (less than 4) of this extreme case of perfect load balance.

By varying the ratio between S_{comp} and S_{comm}, we can alter the bottleneck from computation to communication. A high ratio implies that computation is less likely to be the bottleneck as the compute nodes can process the computation load faster than the links can move the data around. Our results show that our framework provides good trade-offs between the two extremes as the ratio between S_{comp} and S_{comm} is varied. As the ratio tends to zero, our algorithm tends to create very few partitions with very small weight edges across them. On the other hand, as the ratio tends to infinity, the partitioning solution tends to produce N_P partitions that are almost perfectly balanced. Note that another major advantage of our approach is that it need not use all the processors in the system, but can determine if grouping the computation load into bigger and fewer partitions to alleviate the communication bottleneck provides for better throughput.

Our experiments also show that for our simulated streaming applications, our iterative procedure for updating the topology and the partitioning does manage to provide some robustness to the partitioning solution. While for the computation bottleneck, the initial topology and our computation of partitioning from scratch for the computed initial topology already provides good results, our iterative updates manage to improve the throughput for the cases with communication bottleneck.

We now explain our experiments in detail, in particular the characteristics of the synthetic application graphs we used and our validation approach to gain confidence in our results. We discuss both of these issues next.

B. Key Characteristics of Graphs Used in Experiments

While our TPR co-optimizing framework does not depend on a particular class of applications, we performed our experiments for stream computing applications. We picked this because reconfigurable optical circuit switches are particularly suitable for stream computing applications as these applications generate long duration flows that easily compensate for the long (in millisecs) time needed to reconfigure the optical switches. In addition, such applications also benefit the most from the circuit switching (as opposed to packet switching) offered by optical switches.

For simulating stream computing applications, we used a graph generator that we wrote specifically for this purpose. The details of the generation process and an experimental study for evaluating how well the properties of streaming graphs are emulated by our generated graphs is available in a research report [2]. In this section, we discuss the key properties of the graphs we have used for our experiments. This should give the reader specific information to help her interpret our results. However, as we stated before, our proposed techniques do not depend on a particular class of applications.

Based on various discussions with researchers who have first-hand experience with streaming applications, we conjecture that most streaming graphs will satisfy the following properties. However, at the moment, the pool of available streaming applications is too small to obtain statistically relevant measurements supporting these properties. A subset of these properties have also been noted in a recent paper [1]. The statistics on kernel mixes are based on the characterization of StreamIT benchmark [23], [22].

- Streaming graphs are very sparse. Since each edge represents a high-volume, continuous data-flow, a large number of edges imply communication of massive amount of data. This is quite likely a result of a poor design choice.
- There are no vertices with more than 1 in-degree and more than 1 out-degree. All vertices fall into one of the three types:
  - Filters: Vertices with in-degree 1 and out-degree 1. In general, filters can do any kind of data transformation including but not restricted to sampling, filtering, sliding window computations. A special case of identity filters merely pass the data as they receive it.
  - Split: Vertices with in-degree 1 and out-degree greater than 1. The splits are subdivided into following categories: Copy splits that copy the input stream to output streams; Round-robin or If-else distributors that distribute the input stream into output streams.
  - Join: Vertices with in-degree greater than 1 and out-degree 1
- A large majority of the vertices are filters. Around 35% of splits are copy splits – they copy the input stream to output streams. The remaining splits are mostly distributing splits – they distribute the input stream (in a round-robin way, based on value of elements etc.) into the output streams. Most joins merge the input streams in some way, i.e., their output stream rate is the sum of input stream rates.
- Streaming graphs are mostly acyclic. Note that there may be cycles involving control signals that are short data-flows, but we do not model them as edges in our computational task graph. The edges in our definition of streaming graphs strictly correspond to continuous, high-volume data streams.
- For any vertex pair (x, y), all paths from x to y have roughly the same length, where the length is defined as the number of edges in the path independent of the weights on the constituent edges.
- In a typical stream computing application, computationally intensive tasks are usually performed towards the end of the computation process after the initial kernels have sampled and reduced the data volume significantly. In other words, the weights on nodes closer to the sink are significantly higher than those that are closer to the source. On the other hand, the weight of edges that reflect the data-flow volume, decreases as we traverse from sources to sink in the directed acyclic graphs. In particular, the filter vertices in the early part of the computation process significantly reduce the data rate.
- There are more splits than joins close to the sources and there are more joins than splits closer to the sinks.

Our graph generator takes only one input parameter: the number of vertices. The number of edges, and their placement, are determined by the graph generator keeping into account the above properties of stream computing applications.

C. Validation Procedure

We characterize our experiments with the following parameters: N_{nodes}, N_{vertices}, $\Delta_{\max}$, $E_{\max}$, S_{comp}, and S_{comm}, where $\underline{N_{\text{nodes}}}$ is the number of compute nodes in the topology and $\underline{N_{\text{vertices}}}$ is the number of vertices in the application graph generated using Section IV-B. Let a <u>trial</u> be defined as one execution of the sequence "generate an application graph of N_{vertices} vertices, create an initial topology for that on N_{nodes} nodes assuming constraints implied by $\Delta_{\max}$ and $E_{\max}$, and iterate as given in earlier sections to optimize the throughput given the system constraints implied by S_{comp} and S_{comm}." Two trials differ from each other only in the random numbers used to seed the graph generator and the graph partitioner. That is, for each new trial we create a new random application graph, and also seed both METIS and our own algorithm with new seeds. This ensures that our results do not depend on a "lucky" selection of application graph or the parameters for the search algorithms. To further ensure that serendipity is not a factor in the goodness of our reported results, we consider an experiment completed only when enough trials have been performed to give us a certain level of confidence in the results we report. Let the *imprecision*, $\pi(\overline{x})$, of a set of values of x be defined as the half-width of the 95% confidence interval divided by the mean of x. We call an experiment completed when enough trials have been performed to give an imprecision of no more than 5% for the measured throughput.

Output of a given trial gives the following information: the maximum throughput achieved; i_{opt}, the iteration in which the maximum throughput was achieved; Φ, the improvement, if any, over the throughput achieved in the initial topology; and an indication whether this particular value of system throughput was limited by the processing speed of the compute nodes or by the bandwidth of the communication links.

At the end of an experiment, we compute the following additional metrics: $N^{\text{pb}}_{\text{trials}}$, the fraction of total trials that had computation as the bottleneck ('pb' stands for 'processor bottleneck'); $\bar{\Phi}$, the average value for Φ; $\pi(\bar{\Phi})$, the imprecision for $\bar{\Phi}$over N_{trials}trials; and $N^{\text{nc}}_{\text{trials}}$, the fraction of trials that showed no improvement over the initial topology ('nc' stands for 'no change').

D. Detailed Results

We report our results in Table I and Table II, for several experiments, each with a different set of values for parameters N_{nodes}, N_{vertices}, $\Delta_{\max}$, $E_{\max}$, S_{comp}, and S_{comm}. In Table I, the notation $\underline{r_{\text{looseub}}}$ refers to the ratio of actual throughput to the throughput for the hypothetical scenario with perfect load balance on a topology with all-pair links of infinite bandwidth.

Table I shows results for some settings in which the computation is the bottleneck (as shown by high values of $N^{\text{pb}}_{\text{trials}}$). For these settings, the throughput of our partitioning solution is within a small constant factor (at least 27.9%) of the throughput of the extreme case of perfect load-balance.

A clear trend emerges from Table II and Figure 2. As the fraction of total trials that have computation as the bottleneck decreases, the performance improvement given by our algorithm increases. This phenomenon produces three distinct clusters of data. We have separated them in Table II with horizontal lines, and can also be seen in Figure II. The cluster of points with the highest throughput was observed in experiments where most of the trials had communication as the bottleneck, i.e., S_{comp} and S_{comm} were set to give very small values of $N^{\text{pb}}_{\text{trials}}$. Similarly, the cluster of points with the lowest throughput was seen in experiments where most

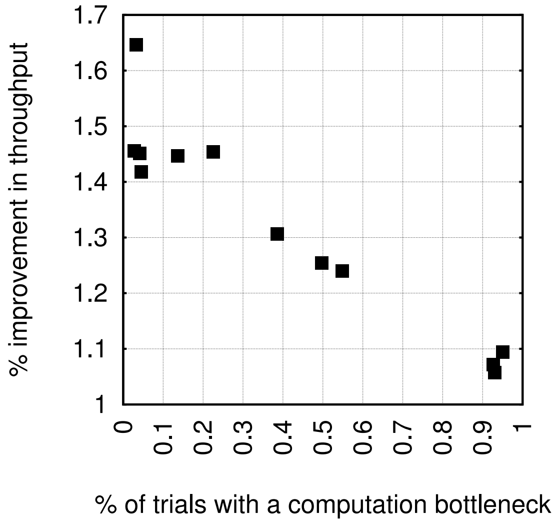

Figure 2. As the fraction of total trials that have computation as the bottleneck increases, the performance improvement given by our algorithm increases.

of the trials had computation as the bottleneck, i.e., S_{comp} and S_{comm} were set to give very high values of $N^{\text{pb}}_{\text{trials}}$.

The fact that our scheme does not show much improvement over the initial topology for high values of $N^{\text{pb}}_{\text{trials}}$is understandable. Recall that the initial topology is based on a condensed graph from METIS partitioning that puts greater emphasis on balancing the load (as maximum allowed load imbalance is 1.05). Therefore, the initial partitioning (computed from scratch afterwards) ensures that the computational load is still well-distributed. Our experiments confirm that computational load is indeed well-balanced after the partitioning for the initial topology and if the computation is the bottleneck, it already provides for good partitioning solution (e.g., for settings in Table I).

Since METIS minimizes the total cut value and the maximum total weight leaving a node (maximum subdomain degree), but not the maximum communication between compute-node pairs, the initial topology may not be as good when communication is the bottleneck. In such cases, our iterative procedure for updating the topology and the partitioning provides significant improvements.

The largest performance improvement we saw was 65% improvement in throughput over that given by the initial topology. This was for the 12th system configuration in Table II. Throughput in this particular configuration was highly limited by the bandwidth available over links (only 3.3% of 360 trials showed computation as a bottleneck). Another point to note here is that this configuration saw one of the smallest values for $N^{\text{nc}}_{\text{trials}}$at 13.3%. That is, our algorithm succeeded in improving the throughput over that of initial topology for almost 87% of all trials.

Although the results shown in this section are for input graphs with small values of N_{vertices} and small number

Table I
EVALUATION AGAINST THE HYPOTHETICAL EXTREME CASE OF PERFECT LOAD BALANCE ON INFINITE BANDWIDTH PERFECTLY CONNECTED NETWORK

	Φ	N_{nodes}	$\Delta_{\max}$	$E_{\max}$	S_{comp}	S_{comm}	N_{vertices}	$N^{\text{pb}}_{\text{trials}}$	$\pi(\Phi)$	N_{trials}	$N^{\text{nc}}_{\text{trials}}$	r_{looseub}
1	1.074	16	4	32	100	500	100	1.000	0.009	252	0.282	0.279
2	1.040	16	4	32	100	500	200	0.991	0.004	348	0.244	0.333
3	1.029	16	4	32	100	500	300	0.996	0.004	240	0.163	0.388

Table II
PARAMETERS FOR THE VARIOUS EXPERIMENTS WE CONDUCTED AND THE CORRESPONDING RESULTS.

	Φ	N_{nodes}	$\Delta_{\max}$	$E_{\max}$	S_{comp}	S_{comm}	N_{vertices}	$N^{\text{pb}}_{\text{trials}}$	$\pi(\Phi)$	N_{trials}	$N^{\text{nc}}_{\text{trials}}$
1	1.306	16	4	32	100	10	100	0.386	0.036	360	0.219
2	1.254	16	4	32	100	10	300	0.497	0.027	720	0.189
3	1.240	16	6	48	100	10	300	0.548	0.048	252	0.214
4	1.094	16	4	32	100	100	100	0.950	0.019	180	0.328
5	1.072	16	4	32	100	100	200	0.927	0.014	648	0.230
6	1.057	16	4	32	100	100	300	0.931	0.013	720	0.185
7	1.454	32	4	64	100	10	300	0.225	0.046	324	0.114
8	1.451	16	4	32	1000	10	200	0.042	0.049	168	0.214
9	1.456	16	4	32	1000	10	300	0.028	0.030	612	0.196
10	1.447	16	4	32	1000	10	1000	0.137	0.030	1253	0.229
11	1.418	16	6	48	1000	10	300	0.045	0.054	156	0.237
12	1.646	32	4	64	1000	10	1000	0.033	0.046	360	0.133

of processors N_{nodes}, our preliminary experiments suggest that similar trends continue for the larger graphs (e.g., with $N_{\text{vertices}} = 64000$) and for larger numbers of processors.

Figure 3 shows a typical scenario for the number of iteration in which the best throughput was computed. In a large majority of our experiments, the best throughput was obtained in very few iterations (less than 3). However, in some cases, we even obtained improvements in iteration 15 or higher.

The time taken by TPR co-optimizing framework to execute can be tuned by a parameter controlling the local search depth in the refinement phase. For a system with 32 nodes and 10,000 vertices, the average time was around 3 minutes.

V. RELATED WORK

We are not aware of any work that configures a topology to match the communication requirements of a given application. We are also not aware of any work that co-optimizes the partitioning and routing. There is however considerable work on computing a partitioning of the input graph for a given architecture, both serial and parallel, and static and dynamic (e.g., [12], [4], [5], [13], [16], [21]). In the context of heterogeneous architectures that have compute nodes with variable processing speed and links with variable bandwidth (such as grid infrastructures), the problem is also referred to as architecture-aware partitioning. This problem is considered difficult even for designing efficient problem-

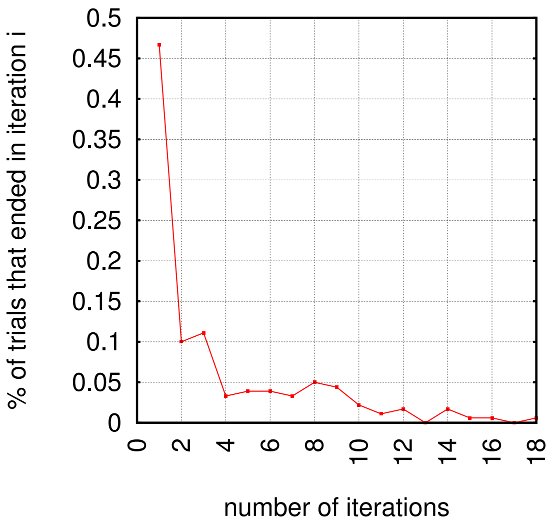

Figure 3. The number of iteration in which the best throughput was computed

specific heuristic approaches. As such, generic local search heuristics have been used for solving this problem, such as genetic algorithms [9] and cross-entropy methods [18]. The few problem-specific heuristics include MinEx [6], MiniMax [14] and a variant of METIS [15].

Our subproblem of computing a topology-aware partitioning (within the host of problems solved under the TPR co-optimizing framework) is actually simpler than the partitioning for completely heterogeneous architecture as we assume that the compute nodes are identical and that the links between them have the same bandwidth. We, however, believe that the technique developed in this paper will also be effective for fully heterogeneous architectures such as those from the domain of grid computing.

The work in [10] is very much related to ours, in the sense that it substantiates our contribution. Our contribution almost starts where the contribution of [10] finishes. Specifically, an important contribution of [10] is doing a very detailed study of different applications used in high performance computing, and showing that their communication requirements do not need as powerful a network as a fat tree. They argue that a fat tree can be pruned into a more economical fit tree. Based on their analysis of communication requirements of different applications, they make an excellent case for the potential of utilizing a reconfigurable interconnect. This study, and some others like it, motivate our present paper on how to utilize a reconfigurable interconnect.

Another closely related technique is the MiniMax approach ([14]) as that technique is also multilevel. But our approach differs in the way it coarsens the graph, it does the initial partitioning, performs refinement, as well as in the objective function itself.

Unlike some previous efforts (e.g., [3]) that have investigated the use of reconfigurable interconnect in high performance computing, our framework does not focus on, or assume, a certain class of applications or certain communication patterns.

An approach for generating task graphs that emulate the key characteristics of streaming applications has recently been proposed [1]. However, this work heavily relies on application-specific input from the user that guides the generation process.

VI. Conclusions and Future Work

In this paper, we have proposed new algorithms for iteratively co-optimizing the network topology configuration, application partitioning and network flow routing to maximize throughput for a given application for systems that employ reconfigurable interconnects.

We show the value of our TPR co-optimizing framework on synthetic graphs that emulate the key characteristics of stream computing applications that require high throughput. We perform statistically rigorous, serendipity-free experiments to show that (a) TPR co-optimizing framework achieves a throughput within a small constant factor (less than 4) of a loose upper bound calculated for a system with no communication bottleneck (b) TPR co-optimizing framework consistently improves (ranging from 42% to 65%) upon the initial throughput for the more relevant cases of communication bound systems.

Our work can be extended to the dynamic setting, where the weights on vertices and edges can change over time. This can be done by an iteration of modifying the topology and updating the partitioning as is done in our framework. However, the current partitioning update procedure re-computes the partitioning from scratch. Ignoring the previous partitioning allows our framework to consider radically different partitioning solutions and may lead to better throughput. But it might not be desirable in a dynamic scenario as this is a slow procedure and more importantly, it might involve a large data-migration between the processors. The cost of data-migration may be more than the actual benefit accrued from re-balancing the load. Therefore, we would like to investigate techniques that use the partitioning from the previous iterations to compute a new partitioning efficiently such that the data-migration requirement between the two partitioning solutions is low.

Another direction of future research is to improve the scalability of the above approach. This may include parallelization, either using MPI interface with threads or using distributed paradigms such as map-reduce and/or making it more cache-efficient.

Our eventual goal is to incorporate this framework in a high-performance computing system with a reconfigurable switch. Given an application with dynamically changing weights on the task graph (possibly because of imprecise estimates on the work load), such a system will decide at the run-time when and how to re-balance the load, re-route the network flows, and re-compute and reconfigure the topology for the best overall throughput. This system will help make the power-efficiency and bandwidth goals of future Exascale systems a reality.

Acknowledgements

The authors are grateful to Kostas Katrinis, Rolf Riesen, Dilma da Silva and Cheng-Hong Li for their valuable feedback. In particular, they helped us build a better understanding of the optical circuit switches.

References

[1] D. Ajwani, S. Ali, K. Katrinis, C.-H. Li, A. J. Park, J. P. Morrison, and E. Schenfeld. A flexible workload generator for simulating stream computing systems. In *Proceedings of the 19th Annual IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 409–417. IEEE, 2011.

[2] D. Ajwani, S. Ali, and J. P. Morrison. Application-agnostic generation of synthetic task graphs for stream computing applications. Technical Report RC25181 (D1107-003), IBM Research, 2011.

[3] K. J. Barker, A. F. Benner, R. R. Hoare, A. Hoisie, A. K. Jones, D. J. Kerbyson, D. Li, R. G. Melhem, R. Rajamony, E. Schenfeld, S. Shao, C. B. Stunkel, and P. Walker. On the feasibility of optical circuit switching for high performance computing systems. In *Proceedings of the ACM/IEEE Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, page 16. IEEE Computer Society, 2005.

[4] U.V. Catalyurek, E.G. Boman, K.D. Devine, D. Bozdag, R. Heaphy, and L.A. Riesen. Hypergraph-based dynamic load balancing for adaptive scientific computations. In *Proceedings of the 21st IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2007)*, 2007.

[5] U.V. Catalyurek, E.G. Boman, K.D. Devine, D. Bozdag, R. Heaphy, and L.A. Riesen. A repartitioning hypergraph model for dynamic load balancing. *Journal of Parallel and Distributed Computing*, 69(8):711–724, 2009.

[6] D. J. Harvey, S. K. Das, and R. Biswas. Design and performance of a heterogeneous grid partitioner. *Algorithmica*, 45(3):509–530, 2006.

[7] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (CDROM)*, Supercomputing '95, New York, NY, USA, 1995. ACM.

[8] M. Holtgrewe, P. Sanders, and C. Schulz. Engineering a scalable high quality graph partitioner. In *Proceedings of the 24th IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2010)*, 2010.

[9] A. Jain, S. Sanyal, S. K. Das, and R. Biswas. Fastmap: A distributed scheme for mapping large scale applications onto computational grids. In *Proceedings of the 2nd International Workshop on Challenges of Large Applications in Distributed Environments (CLADE)*, page 118. IEEE Computer Society, 2004.

[10] S. Kamil, L. Oliker, A. Pinar and J. Shalf, "Communication Requirements and Interconnect Optimization for High-End Scientific Applications," IEEE Transactions on Parallel and Distributed Systems (TPDS), 2009 . Communication Requirements and Interconnect Optimization for High-End Scientific Applications *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 21(2):188–202, 2009.

[11] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *Journal on Scientific Computing*, 20(1):359 – 392, 1999.

[12] G. Karypis and V. Kumar. Parallel multilevel series k-way partitioning scheme for irregular graphs. *SIAM Review*, 41(2):278–300, 1999.

[13] R. Khandekar, K. Hildrum, S. Parekh, D. Rajan, J. Wolf, K.-L. Wu, H. Andrade, and B. Gedik. COLA: optimizing stream processing applications via graph partitioning. In *Middleware '09: Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, 2009.

[14] S. Kumar, S. K. Das, and R. Biswas. Graph partitioning for parallel applications in heterogeneous grid environments. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS 2002)*, 2002.

[15] I. Moulitsas and G. Karypis. Architecture aware partitioning algorithms. In *Proceedings of the 8th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, Vol. 5022 of *Lecture Notes in Computer Science*, pages 42–53. Springer, 2008.

[16] F. Pellegrini. Contributions au partitionnement de graphes parallle multi-niveaux / contributions to parallel multilevel graph partitioning. *LaBRI, Universit Bordeaux*, 2009.

[17] P. Pepeljugoski, J. Kash, F. Doany, D. Kuchta, L. Schares, C. Schow, M. Taubenblatt, B. Offrein, and A. Benner. Towards exaflop servers and supercomputers: The roadmap for lower power and higher density optical interconnects. In *Proceedings of the 36th European Conference on Optical Communications (ECOC)*, 2010.

[18] S. Sanyal and S. K. Das. Match : Mapping data-parallel tasks on a heterogeneous computing platform using the cross-entropy heuristic. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS 2005)*, 2005.

[19] L. Schares, X. J. Zhang, R. Wagle, D. Rajan, P. Selo, S.-P. Chang, J. R. Giles, K. Hildrum, D. M. Kuchta, J. L. Wolf, and E. Schenfeld. A reconfigurable interconnect fabric with optical circuit switch and software optimizer for stream computing systems. In *Optical Fiber Communication Conference*, 2009.

[20] N. Selvakkumaran and G. Karypis. Multiobjective hypergraph-partitioning algorithms for cut and maximum subdomain-degree minimization. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 25(3):504–517, 2006.

[21] A. J. Soper, C. Walshaw, and M. Cross. A Combined Evolutionary Search and Multilevel Optimisation Approach to Graph Partitioning. *J. Global Optimization*, 29(2):225–241, 2004.

[22] W. Thies. *Language and Compiler Support for Stream Programs*. PhD thesis, Massachusetts Institute Of Technology, 2009.

[23] W. Thies and S. P. Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *Proceedings of the 19th International Conference on Parallel Architecture and Compilation Techniques (PACT 2010)*, 2010.

[24] C. Walshaw and M. Cross. Parallel Optimisation Algorithms for Multilevel Mesh Partitioning. *Parallel Computing*, 26(12):1635–1660, 2000.