Research Repository UCD

Title	Co-optimizing application partitioning and network topology for a reconfigurable interconnect
Authors(s)	Ajwani, Deepak, Hackett, Adam, Ali, Shoukat
Publication date	2016-10
Publication information	Ajwani, Deepak, Adam Hackett, and Shoukat Ali. "Co-Optimizing Application Partitioning and Network Topology for a Reconfigurable Interconnect." Elsevier, October 2016. https://doi.org/10.1016/j.jpdc.2016.04.010.
Publisher	Elsevier
Item record/more information	http://hdl.handle.net/10197/9893
Publisher's statement	This is the author's version of a work that was accepted for publication in Journal of Parallel and Distributed Computing. Changes resulting from the publishing process, such as peer review, editing, corrections, structural formatting, and other quality control mechanisms may not be reflected in this document. Changes may have been made to this work since it was submitted for publication. A definitive version was subsequently published in Journal of Parallel and Distributed Computing (96 (2016)) https://doi.org/10.1016/j.jpdc.2016.04.010
Publisher's version (DOI)	10.1016/j.jpdc.2016.04.010

Downloaded 2026-05-01 23:41:40

The UCD community has made this article openly available. Please share how this access benefits you. Your story matters! (@ucd_oa)

© Some rights reserved. For more information

Co-optimizing Application Partitioning and Network Topology for a Reconfigurable Interconnect

Deepak Ajwani[a,*], Adam Hackett[b], Shoukat Ali[c], John P. Morrison[d], Stephen Kirkland[b,**]

[a] *Bell Labs, Alcatel-Lucent, Dublin 15, Ireland*
[b] *Hamilton Institute, National University of Ireland Maynooth, Ireland*
[c] *Elastica, San Jose, CA*
[d] *The Centre for Unified Computing, University College Cork, Cork, Ireland*

*Corresponding author: Email deepak.ajwani@alcatel-lucent.com, Phone +353-1-8864709.
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). Earlier versions of portions of this research appeared in [1, 2].
**Current address: Department of Mathematics, University of Manitoba, Winnipeg, Manitoba, Canada

Abstract

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. Finally, we show how using a particular topology as a seed to our framework significantly reduces the time to compute the final topology.

1. 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., [4, 5, 6, 7]). 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 and routing of G on the interconnect topology?" 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 TPR co-optimizing framework.

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 2. Sec-

tion 3 describes our main framework together with all the details of the individual steps. Our experiments with this framework are reported in Section 4. In Section 3.2, we show that seeding the framework with a good initial topology and then conducting a restricted search around it significantly reduces the time to compute the final topology. We describe some related work in Section 5 and conclude with future research directions in Section 6.

2. Preliminaries

2.1. 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 vertices while elements in 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 $\underline{w_e(e)}$.

2.2. 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 3.3.3).

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. (a) Compute a network topology graph H that is likely to elicit a high throughput mapping for the application graph G. (b) Compute a mapping of vertices in V_G to nodes in V_H to achieve a high computation throughput. (c) 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.

2.3. Key Contributions

Our key contributions are as follows.

(1) We propose a new framework that iteratively co-optimizes the discovery of a good network topology that is configurable within the constraints of the reconfigurable interconnect, and the computation of a good partitioning, mapping and routing solution for deploying the application on the computed topology to maximize throughput. In contrast to the existing literature on graph partitioning, mapping and routing, our problem is challenging because we are also optimizing for the topology while computing the mapping of an application graph to the

topology. Thus in our framework, the topology is iteratively altered to alleviate the computation and communication bottlenecks that are identified with the mapping solution.

As an important part of our framework, we propose a heuristic for architecture-aware partitioning and high-throughput routing. This heuristic does not assume a fixed topology, but can work with arbitrary topologies configurable by a reconfigurable interconnect.

(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) We explore the use of chordal graph as the seed topology in our iterative framework.

(4) An extensive empirical analysis to show that (i) our initial topology yields a high throughput partitioning and routing solution in many cases, (ii) our iterative co-optimization framework provides robustness to the initial topology by modifying the initial topology in a way that yields better partitioning and routing solutions and (iii) our framework results in high throughput solutions, irrespective of whether computation or communication is the bottleneck.

2.4. 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.

3. 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, a 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: (a) Modify the network topology based on the existing partitioning and routing of the application graph. (b) 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. As most variants of graph partitioning (including many variants of the topology-aware partitioning and mapping problems) and low-congestion routing problems are NP-hard (see e.g. [8, 9]), we need heuristics to solve them efficiently. We now present our heuristics for engineering a good solution for the TPR co-optimization.

3.1. Computing Initial Topology

A fundamental question that we address in this section is given an application graph, what is a topology that will yield a high throughput partitioning and routing solution. 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 this approach to be effective and this approach may not be scalable.

Therefore, we use an alternative approach based on condensed graph to determine a good initial topology. The basic idea behind our approach is to first compute an architecture-oblivious partitioning of the application graph, condense the application graph using this partitioning and then fix the condensed graph to satisfy the constraint on maximum degree and the total number of edges. Figure 1 illustrates some of the steps involved in creating the initial topology. Please refer to that as needed.

Later in Section 3.2, we explore initial topologies based on chordal rings.

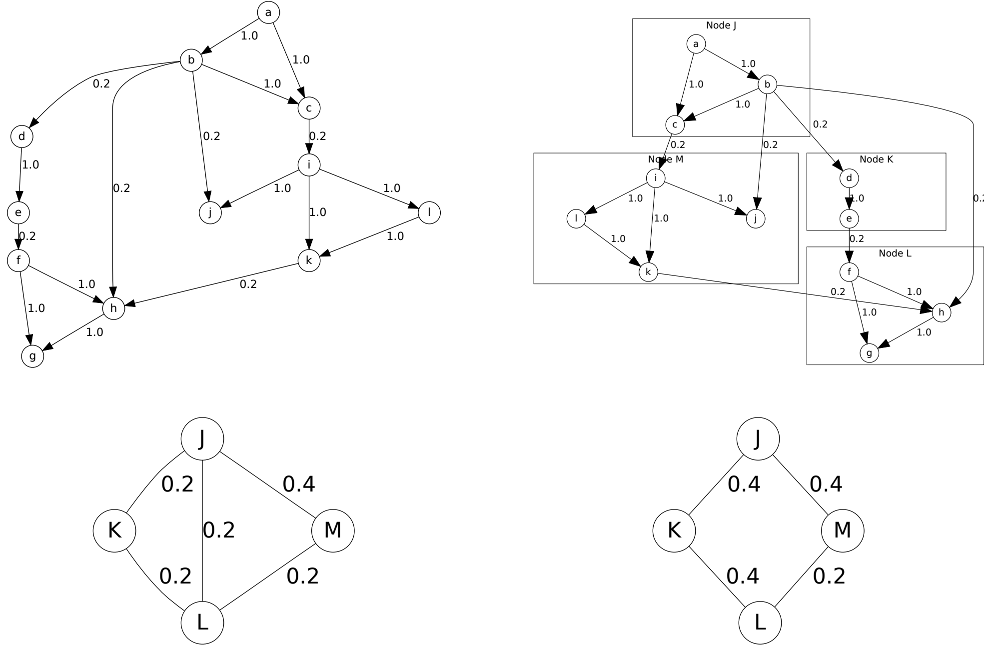

Figure 1: This figure shows various steps in computing an initial topology of the application graph shown in top left for an architecture with 4 compute nodes, a max-degree constraint of at most two physical communication links per compute node, and a max-edges constraint of at most 4 physical communication 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. Note that even though the application graph is directed, the initial topology computed is undirected because of the bi-directional nature of the links.

3.1.1. 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.

1. 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.

2. 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.

3. 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.

In order to obtain the initial topology, we want to later fix the maximum degree of the condensed graph resulting from this partitioning. Ideally, we want to perform minimal structural changes in the condensed graph and as such, we would like the condensed graph to have it's maximum (unweighted) degree either less than or as close as $\underline{\Delta_{\max}}$ (the constraint on the maximum degree of the interconnect topology) as possible. The third objective was chosen in the hope that the weighted degree of a subdomain is correlated to the unweighted degree of the node corresponding to the subdomain in the condensed graph.

We use the graph partitioning library METIS [10] 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 [11]. (While we chose to use METIS for this research, other graph partitioning software could have been used as well, e.g., Chaco [12], JOSTLE [13], Scotch [14] and Zoltan [15]).

3.1.2. Creating Connected Condensed Graphs

Based on the topology-oblivious partitioning, we condense the input graph G to form a condensed graph G_c. 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 with weight 0.

3.1.3. 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 (shown as e in Figure 2a) and let the graph be disconnected into two components (P and Q in Figure 2b). We pick the minimum weight edges $\{u, v\}$ and $\{w, x\}$ in the two components (Figure 2c), remove them and insert

$\{u, w\}$ and $\{v, x\}$ (Figure 2d). This "edge swap" 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.

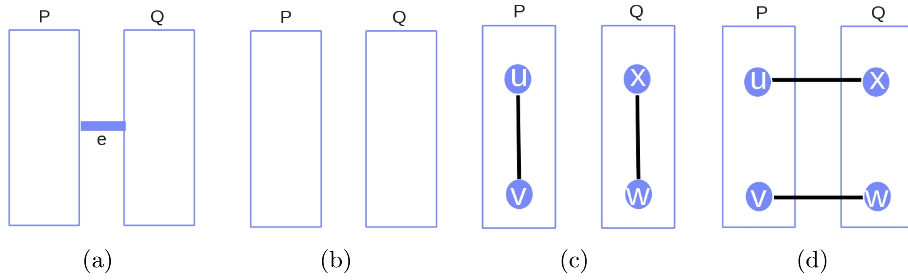

Figure 2: The edge swap transformation.

3.1.4. 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.

3.2. Chordal Ring Initial Topology

A major shortcoming of the condensed graph based initial topology computation technique is that there is no purpose-built capacity in the resultant topology to handle link congestion. A poor partitioning or the subsequent alterations in the topology can lead to a topology that does not mimic the communication pattern of the application properly and thus, result in poor communication throughput.

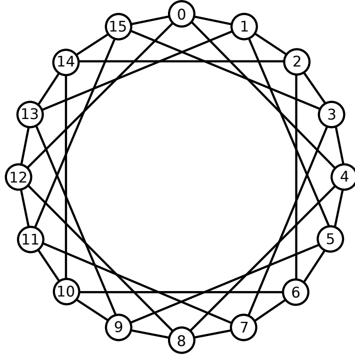

Figure 3: The chordal ring $C(16; 4)$.

To overcome this shortcoming, we propose an alternative framework. Rather than starting with a condensed graph, we initialize our topology search with a chordal ring topology. A chordal ring is a k-regular circulant graph formed by the addition of edges between pairs of vertices at specified distances along the ring. More precisely, a ring with vertices labelled $i = 0, \ldots, n-1$ and an associated set of chord lengths, $Q \subseteq \{q_1, q_2, \ldots, q_{\lfloor n/2 \rfloor}\}$, forms the chordal ring $C(n; q_1, \ldots, q_{|Q|})$, if for each $q_j \in Q$ there is an edge from vertex i to vertex $(i + q_j) \mod n$. The degree of the graph is determined by the choice of Q. In general $k = 2|Q| + 2$ (see Fig. 3). For even values of n, chords of length $n/2$ yield $k = 2|Q| + 1$. Recall that k is the vertex degree in this k-regular graph.

This class of graphs has been used extensively in theoretical studies of interconnect design (e.g., [23, 24, 25]), and it is known to possess a number of structural properties that have traditionally been seen as vital for robustness against node or link failures [26, 27]. Given that the maximum edge constraint enforced by the configurable optical switch may result in removal of some edges from our chosen initial topology (even if it is a condensed graph), it makes sense to start with a topology that is known to be robust against such link failures. *Intentional link removal* can be viewed as a type of link failure. Furthermore, the same characteristics of chordal rings that make them attractive for robustness against failed links also make them attractive for robustness against congested links. In both cases, the structure of the chordal rings makes it likely that many communication paths will remain available even in the face of congestion of certain links. Note that this is not a design feature of the condensed graphs.

We note that other graphs like de Bruijn graphs [28] or Kautz graphs [29] also provide such robustness. However, the chordal rings provide much more

design flexibility in terms of choosing the required number of vertices while also satisfying the maximum degree constraint. The maximum edge constraint can then be satisfied by removing a random subset of edges (while preserving the connectivity).

Consider the case of $\Delta_{\max} = 4$. To satisfy this particular maximum degree constraint, we focus our discussion on simple degree four chordal rings $C(n; q)$ [30], i.e., a ring in which all chords have the same length, q. There is a number of interesting properties unique to this particular subclass. Iwasaki *et al.* [31] have shown that for every graph in $C(n; q)$ there are four independent spanning trees rooted at each vertex. In other words, between every pair of (source, destination) vertices there are four edge-disjoint paths, the maximum possible number for a 4-regular graph.

Another interesting property of $C(n; q)$ is their bisection width. Bisection width is important for our particular problem because a higher bisection width will translate into a larger number of paths between any two bisections, thereby permitting a better congestion alleviation between the two given bisections.

For large values of n, the bisection width of chordal rings is usually better than that of 2-D torus. For instance, the bisection width of $C(2^k; 4)$ is known to be 2^{k-1} [32], suggesting a linear growth as a function of n, while that of 2-D torus is $2n^{1/2}$, a sub-linear function. This suggests that such a chordal ring may be a better candidate for a seed topology than a 2-D torus.

3.3. Partitioning Input Graph for the Given Topology

Next, we focus on partitioning the application graph for a given topology. Our scheme for partitioning the input graph for the given topology is based on a multilevel scheme. Multilevel techniques (e.g., [10, 13, 12, 14]) 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 matchings are uncontracted. After uncontracting a matching, the refinement algorithm uses some local search heuristics to improve the partitioning objective.

The basic assumption behind multilevel partitioning approaches is that the coarsening phase typically preserves the structure of the input graph while reducing its size and therefore, a good partitioning at a coarser level can be used to obtain a good partition at the finer level with some simple refinement heuristics. These approaches have been shown to provide good partitions in very little time on a large range of graphs. Please refer to extensive literature on multilevel partitioning (e.g., [10, 13, 12, 14]) 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.

3.3.1. Coarsening

For the coarsening phase, matchings based on edge expansion have been shown to be more effective than those based on edge weights [16]. 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. Note that this value of number of vertices has been experimentally determined to ensure that we get good performance on a wide range of problem settings. However, our results are not too sensitive on this value.

3.3.2. 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 [14].

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.

3.3.3. 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 $0 \le j < i$. 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: (a) 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. (b) For each edge $e \in E_G$, we consider re-routing it through a minimum-congestion path.

3.4. 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. This is done as follows: Given the partitioning and routing solution computed for the

topology at this stage (as described in the previous section), we compute the computation load $w_v(P)$ on each node P as the sum of compute loads of all application vertices mapped to it. Similarly, the communication load $w_e(e)$ on each link e is calculated by summing over all the flow passing through it (the formal definition has earlier been given in Section 2.2). The maximum computation throughput supported by a node P is $S_{\text{comp}}/w_v(P)$ and the maximum communication throughput supported by an edge e is $S_{\text{comm}}/w_e(e)$. The bottleneck is the compute node or communication link with the smallest supported throughput value. If the bottleneck is a node, we identify it as a computation bottleneck and if it is a link, we identify it as a communication 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 depict this situation in Figure 4. As one solution, we select a light-weight edge between two low-weight nodes (x and z in Figure 4) and a light-weight edge between two high-weight nodes (u and v in Figure 4) 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 $\{z, x\}$ involves removing the two edges and either inserting $\{u, z\}$ and $\{v, x\}$ or inserting $\{u, x\}$ and $\{v, z\}$.

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

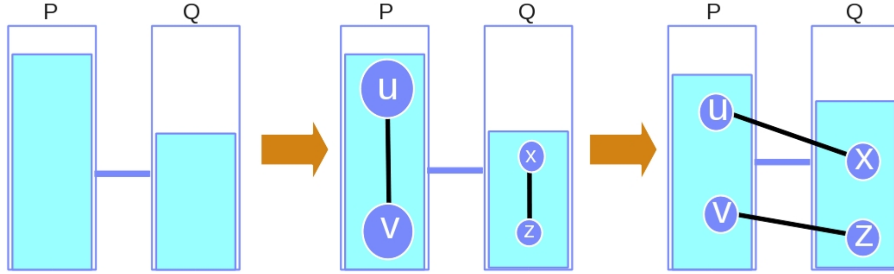

Figure 4: Changing topology to fix a computation bottleneck.

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: (a) The two edges are vertex disjoint. (b) Swapping the edges leaves the graph connected. (c) 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. We depict this process in Figure 5 and explain below.

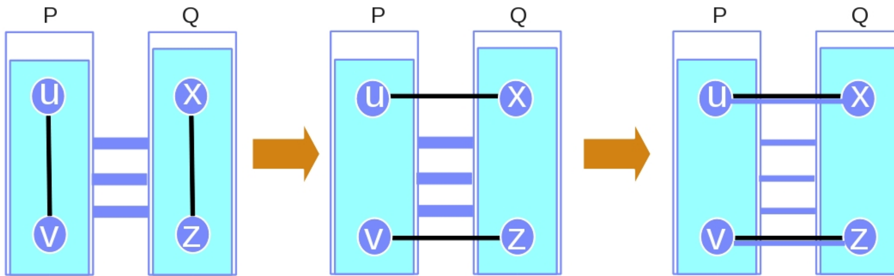

Figure 5: Changing topology to fix a communication bottleneck.

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 (partitions P and Q in Figure 5) are identified, the edges are re-inserted into the graph. We then find low-weight edges from both sides ($\{u, v\}$ from P and $\{x, z\}$ from Q in Figure 5) such that they can be swapped and swap it to obtain H_{i+1} (the graph in the middle in Figure 5). This process has now introduced two more edges between P and Q that can then be used to offload some communication from the congested links between P and Q (as shown in the right-most subfigure in Figure 5). Figure 6 illustrates this process for an example graph in a step-by-step fashion.

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.

4. Experiments and Results

4.1. 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.

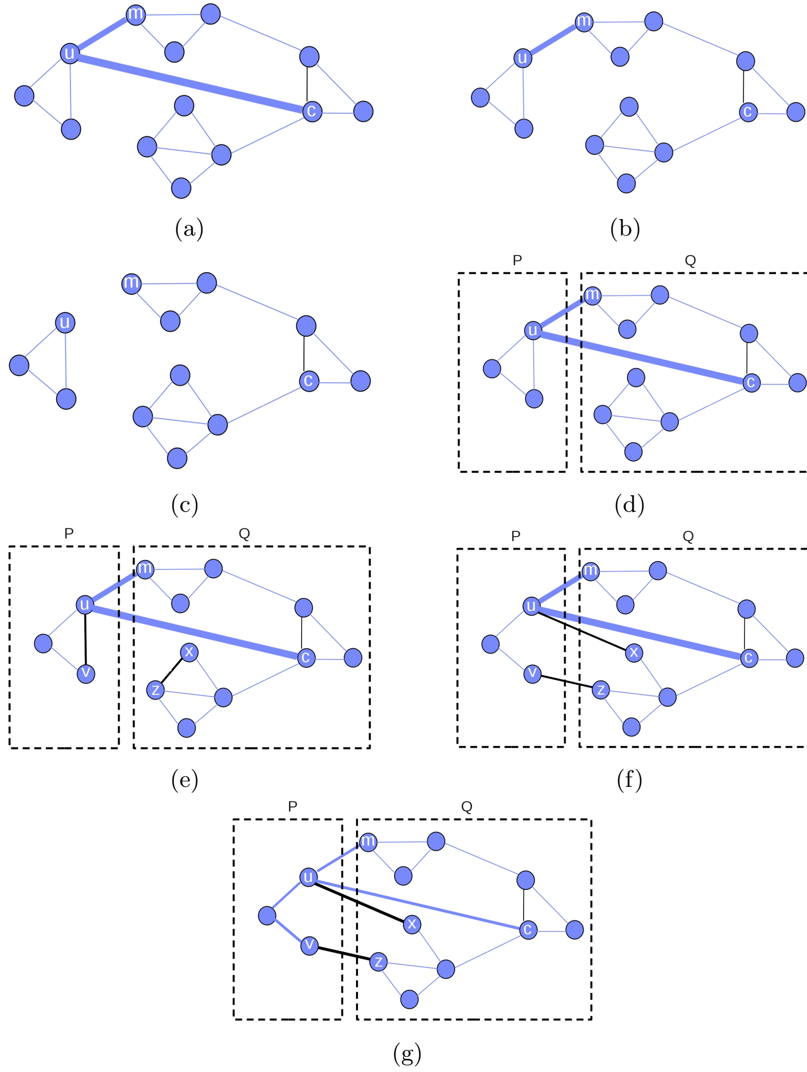

Figure 6: An example of modifying topology to fix a communication bottleneck. (a) A topology with a communication bottleneck due to edges $\{u, m\}$ and $\{u, c\}$. (b) Trying to identify the bottleneck cut by removing the heaviest edge $\{u, c\}$ to see if that disconnects the graph. (c) Removing $\{u, c\}$ did not disconnect the graph, so removing the next heaviest edge $\{u, m\}$. (d) Bottleneck cut identified as that between partitions P and Q. (e) Identified a light edge in each partition, $\{u, v\}$ and $\{x, z\}$. (f) Removed edges $\{u, v\}$ and $\{x, z\}$. Inserted edges $\{u, x\}$ and $\{v, z\}$. (g) Re-routed congestion present on edges $\{u, m\}$ and $\{u, c\}$ through new edges $\{u, x\}$ and $\{v, z\}$.

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.

4.2. Graphs Considered

While our TPR co-optimizing framework does not depend on a particular class of applications, we performed our experiments mostly 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.

Although there are many different applied areas of stream computing such as real-time analysis of financial and medical data [17] and analysis of social networks [18], the pool of *publicly available* mature large real-world stream computing applications is fairly small. Since we need a large number of application graphs of varying sizes to evaluate different settings in our experiments, our requirements can't all be met by the limited number of public datasets. Therefore, we focus on synthetic stream-computing task graphs and in particular, we use a graph generator specifically written for simulating streaming applications [19, 20].

In addition, we also perform experiments with many graphs from well-known graph-partitioning archives [21, 22] to show that our techniques are equally suited for graphs from other domains as well.

4.3. Validation Procedure

Let a trial 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

Table 1: Various parameters used in our experiments and results

N_{nodes}	Number of compute nodes in the topology
N_{vertices}	Number of vertices in the application graph generated using Section 4.2
$\Delta_{\max}$	Maximum permissible degree of a node in the topology graph (c.f. Section 2.2)
$E_{\max}$	Maximum number of edges allowed in the topology graph (c.f. Section 2.2)
S_{comp}	Processing speed of a compute node in the topology (c.f. Section 2.2)
S_{comm}	Bandwidth of a link in the topology (c.f. Section 2.2)
N_{trials}	Number of trials
$N^{\text{pb}}_{\text{trials}}$	Fraction of total trials that had computation as the bottleneck ('pb' stands for 'processor bottleneck')
$N^{\text{nc}}_{\text{trials}}$	Fraction of trials that showed no improvement over the initial topology ('nc' stands for 'no change')
Φ	Improvement, if any, over the throughput achieved in the initial topology
$\bar{\Phi}$	Average value for Φ
$\pi(\bar{\Phi})$	Imprecision for $\bar{\Phi}$ over N_{trials} trials

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*, $\underline{\pi(x)}$, 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.

We characterizing our experiments and the results with the parameters defined in Table 1.

4.4. Detailed Results

We report our results in Table 2 and Table 3, 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 2, 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 2 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

Table 2: Evaluation against the hypothetical extreme case of perfect load balance on infinite bandwidth perfectly connected network

	$\bar{\Phi}$	N_{nodes}	$\Delta_{\max}$	$E_{\max}$	S_{comp}	S_{comm}	N_{vertices}	$N_{\text{trials}}^{\text{pb}}$	$\pi(\bar{\Phi})$	N_{trials}	$N_{\text{trials}}^{\text{nc}}$	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

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.

Table 3: Parameters for the various experiments we conducted and the corresponding results.

	$\bar{\Phi}$	N_{nodes}	$\Delta_{\max}$	$E_{\max}$	S_{comp}	S_{comm}	N_{vertices}	$N_{\text{trials}}^{\text{pb}}$	$\pi(\bar{\Phi})$	N_{trials}	$N_{\text{trials}}^{\text{nc}}$
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

A clear trend emerges from Table 3 and Figure 7a. 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 3 with horizontal lines, and can also be seen in Figure 3. 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 $N_{\text{trials}}^{\text{pb}}$. Similarly, the cluster of points with the lowest throughput was seen in experiments where most of the trials had computation as the bottleneck,

i.e., S_{comp} and S_{comm} were set to give very high values of $N_{\mathrm{trials}}^{\mathrm{pb}}$.

The fact that our scheme does not show much improvement over the initial topology for high values of $N_{\mathrm{trials}}^{\mathrm{pb}}$ 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 2).

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 3. 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 $N_{\mathrm{trials}}^{\mathrm{nc}}$ 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 of processors N_{nodes}, we found that similar trends continue for the larger graphs (e.g., with $N_{\mathrm{vertices}} = 64000$) and for larger numbers of processors.

4.5. Running Time

The running time of our scheme varies significantly depending on whether the bottleneck is on the extreme communication side, extreme computation

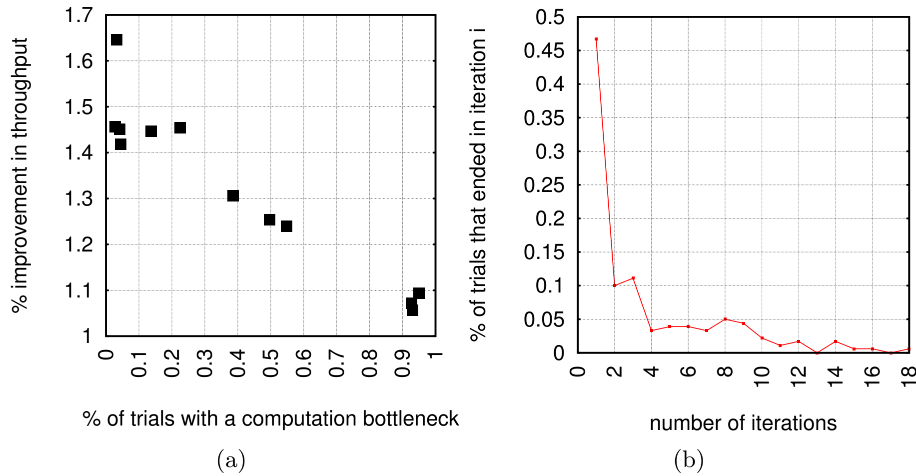

Figure 7: (a) As the fraction of total trials that have computation as the bottleneck increases, the performance improvement given by our algorithm increases. (b) The number of iteration in which the best throughput was computed.

side or it shifts from one to another depending on the selected topology and computed partitioning solution. If the space of good solutions is fairly big or the initial topology is close to a locally optimal solution, our algorithm converges fast requiring very little time. We found that this is indeed a very common case. Figure 7b 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.

Table 4 shows the runtime of our approach on graphs taken from Walshaw's graph-partitioning archives [21] and the 10th DIMACS implementation challenge graph archive [22]. For these experiments, we had used $N_{\text{nodes}} = 16, E_{\max} = 28, S_{\text{comp}} = 500$ and $S_{\text{comm}} = 325$. We find that with these settings, the running time for application graphs with tens of thousands of edges is in a few minutes with a sequential code running on a single core of a typical desktop PC. For instance, on data graph with more than 15K edges, our scheme required less than a minute to compute a 16 node topology and around three minutes to compute a 32 node topology.

Our scheme scales to large application graphs. We were able to compute a high throughput solution for citeseer coauthorship application graph with more than 800K edges in less than 3 hours.

Table 4: Runtime results of our approach on graphs from graph-partitioning arvhives.

	V	E	node degree			Clust. coeff.	Running time
			min	max	avg	avg	(mins)
add20	2,395	7,462	1	123	6.23	0.63	6.6
data	2,851	15,093	3	17	10.59	0.48	0.9
3elt	4,720	13,722	3	9	5.81	0.41	1.9
whitaker3	9,800	28,989	3	8	5.92	0.41	2.2
crack	10,240	30,380	3	9	5.93	0.47	4.2
wing_nodal	10,937	75,488	5	28	13.80	0.42	5.9
fe_4elt2	11,143	32,818	3	12	5.89	0.42	3.5
memplus	17,758	54,196	1	573	6.10	0.77	29.2
coAuthors Citeseer	227,320	814,134	1	1372	7.16	0.68	165.8

The above running time is acceptable for a static scenario where the optimal topology, partitioning and routing solution is computed once in the beginning, the configuration is set and is used for a long running application. This is particularly relevant for many stream-computing applications.

To improve the running time even further (a requirement if the solution needs regular updates and the workload changes frequently), we investigate the runtime deeper. The main components of the running time of our scheme is (i) time to compute the initial topology, (ii) the number of iterations required to converge, (iii) identifying bottleneck link or node and (iv) optimizing the whole performance vector in the refinement phase of computing a topology-aware partitioning and routing solution. Note that the later two points are

repeated for each iteration and hence, the number of iterations also has a bearing on the time spent in those components.

For the first two components, our approach based on chordal ring graphs (described in Section 3.2) reduces the time to compute the initial topology and more importantly, reduces the number of iterations to converge. The third component of identifying the bottleneck is an embarrasingly parallel task as calculating the load on each node and link can be done independently. As for the last component, we can tune the local search depth in the refinement phase by a parameter. Together these optimizations significantly reduce the running time of our approach.

4.6. Experiments with Chordal Ring Initial Topologies

We characterize our experiments with the following parameters: T_{init}, N_{nodes}, N_{vertices}, $\Delta_{\max}$, $E_{\max}$, S_{comp}, and S_{comm}, where $\underline{T_{\text{init}}}$ is the initial topology, Let $\mathcal{T}$ be the set of seed topologies being compared in an experiment. Let a trial be defined as one execution of the sequence "generate an application graph of N_{vertices} vertices, fork $|\mathcal{T}|$ ways, use $\mathcal{T}_i$ as T_{init} in the i-th fork as an initial topology, satisfy constraints implied by $\Delta_{\max}$ and $E_{\max}$, and search the space around this initial topology using the iterative procedure in Section 3." As in Section 4.3, two trials differ from each other only in the random numbers used to seed the application graph generator and the graph partitioner.

We focus on 16-node topologies with $\Delta_{\max} = 4$ and $22 \leq E_{\max} \leq 32$. Note that even for this restricted setting, the number of graphs satisfying the two constraints is quite large and exploring the entire search space (every time we re-configure the switch) is infeasible.

The initial topology of a condensed graph is used as our baseline in this paper. All other topologies are compared to this baseline. Output of a given trial gives the following information for each seed topology T_i: $\underline{\delta_i^{\text{thru}}}$, the ratio of improvement in throughput achieved by T_i over the condensed graph to the throughput achieved by condensed graph, and $\underline{\delta_i^{\text{time}}}$, the ratio of improvement in time to convergence achieved over the condensed graph to time to conver-

gence for topology T_i. At the end of an experiment, we compute the following additional metrics for each seed topology T_i: $\underline{\Delta_i^{\text{thru}}}$, the average value of δ_i^{thru} over all trials and $\underline{\Delta_i^{\text{time}}}$, the average value of δ_i^{time} over all trials.

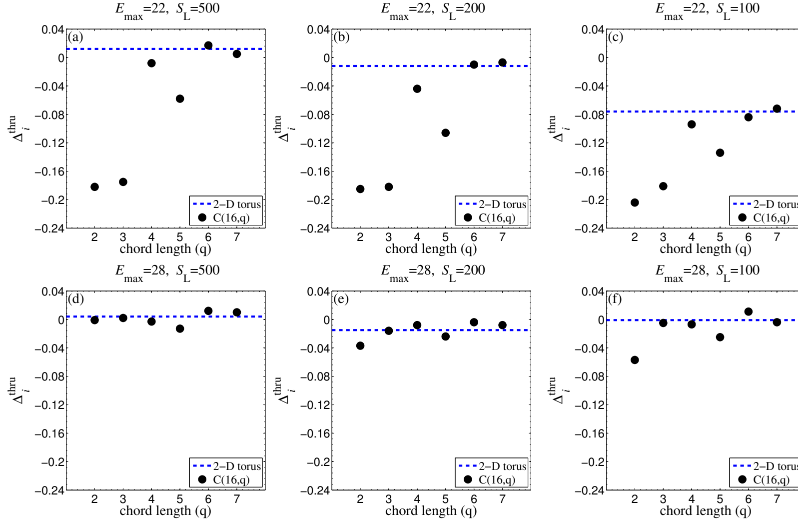

Figure 8: The effect on throughput improvement, Δ_i^{thru}, of changes in the chord length of the 16-node chordal ring.

Figure 8 shows results for a subset of experiments in which only the seed topology, T_{init}, maximum number of allowed edges, E_{max}, and the speed of the network links, S_{comm}, were changed while other parameters were kept at N_{nodes} = 16, N_{vertices} = 300, and Δ_{max} = 4. These results show that, in general, throughput is better if the seed topology is a condensed graph. However, using a suitable chordal ring as a seed topology gave, for most of these experiments, an average throughput that was no worse than 95% of that obtained using the condensed graph, *but* did so in a much shorter time.

There are also some specific insights highlighted in Fig. 8, which shows results for 6 different settings. For easier navigation, the subplots have been arranged so that the experimental set-up becomes more and more challenging communication wise if one moves either to the right or the top. That is, Fig. 8(c) has the most communication wise challenging (hereafter referred to as just ‘chal-

lenging') set-up. This is because (i) the link speed is the lowest, at 100, of three speeds (100, 200 and 500) shown in this figure, and (ii) the number of edges simultaneously allowed in the network is the smaller, at 22, of the two values shown here (22 and 28). Figure 9 has been set up similarly to Fig. 8 except that it shows the effect on convergence time of the choice of initial topology. Notice that any move from a less challenging to a more challenging set-up shows that the rings with higher chord lengths generally do better. Specifically, one can see the following.

(1) A rightward movement from (a) to (b) to (c) in Fig. 8 shows that the number of chordal rings that achieve a throughput that is within 8% of the baseline drops from 4 to 3 to 2.[1] While $C(16;6)$ and $C(16;7)$ reach 92% of the baseline throughput for all three scenarios, $C(16;5)$ drops out of the "well-performing" set of rings when S_{comm} is decreased from 500 to 200. And both $C(16;4)$ and $C(16;5)$ drop off this set when S_{comm} is further decreased from 200 to 100. That is, for subplot (c), the chord lengths of 2, 3, 4, and 5 are all in a set that does not give throughput competitive to that of the condensed graph. A similar change in performance is seen when one moves 'up' in Fig. 8, from (d) to (a), from (e) to (b), or from (f) to (c). For example, when we move from (d) to (a) in Fig. 8 (i.e., decreasing E_{max} to 22 while keeping S_{comm} constant at 500), the rings with the smaller chord lengths, $C(16;2)$ and $C(16;3)$, start performing quite poorly. As a specific example, the throughput improvement worsens by about 18% for $C(16;2)$.

(2) A similar rightward movement in Fig. 9 shows that the improvement in time taken to converge has, in general, gotten better. Note that the savings in the convergence time are quite significant. For example, for Fig. 9(a), the iterative algorithm converged sooner, as compared to the baseline, by about 19% to 31%, depending on the choice of chord length. The corresponding value for 2-D torus was 20%. When we move one step right to Fig. 9(b), the convergence time

[1] The number 8% is chosen here arbitrarily to make our point. It just needs to be a small number.

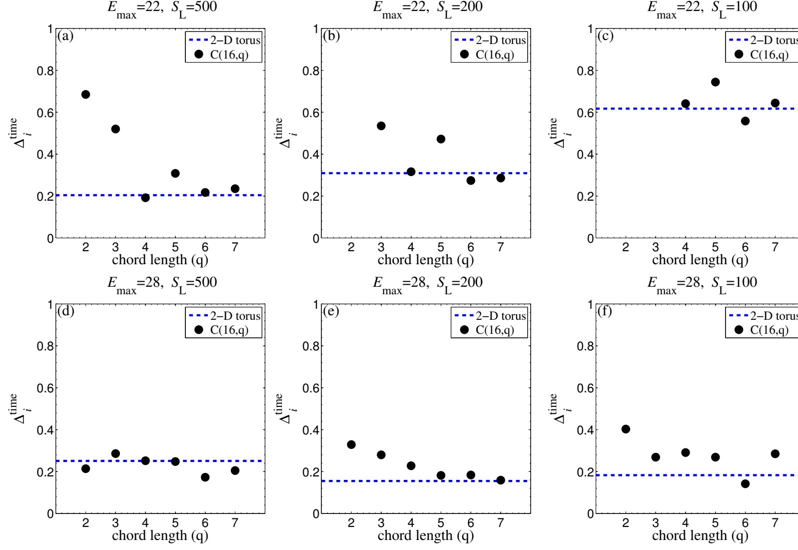

Figure 9: The effect on convergence time, Δ_i^{time}, of changes in the chord length of the 16-node chordal ring.

improvement for the chordal rings gets in the range of 28% to 31%; for 2-D torus, 31%. Note that the apparent much higher improvements, 116% (not shown to reduce congestion on y-axis) and 54%, in convergence times for $C(16;2)$, and $C(16;3)$, respectively, are meaningless here because their performance was worse by more than 8% of the baseline. After another step to the right, in Fig. 9(c), we see time improvements of 51% and 68% for the only two rings, $C(16;6)$ and $C(16;7)$, respectively, that have survived falling off the so-called "well-performing" set. The 2-D torus saves 55% in time in this case.

(3) The 2-D torus as an initial topology works just well as an appropriate chordal ring as far as the throughput improvement is concerned. However, for savings in convergence time, an appropriate chordal ring may be better (see Fig. 9(a), (e), and (f)).

One explanation of the above effect of larger chord length chordal rings "holding out" longer in the face of increasingly adverse communication environments is offered by the average shortest path length, L_{asp}, of chordal rings (Fig. 10). The number of links in the topology path that are loaded to route

a given application edge is smaller for a chordal ring with a smaller L_{asp}. This reduces the maximum load on a given topology link, and subsequently increases the communication throughput. Also, as the links are removed from the topology to satisfy the constraint on the total number of edges, the load on the removed links has to be re-routed through another path. For a topology with a smaller L_{asp} value, fewer links get loaded in the process of re-routing and this results in less congestion overall.

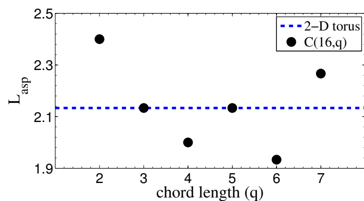

Figure 10: L_{asp} against chord length

As Fig. 10 shows, the L_{asp} value decreases from chord length 2 to 6, except for chord length of 5. This pattern is consistent with the throughput increase pattern shown in Fig. 8. Note that this consistency between the throughput pattern and average shortest path is more prominent for $E_{\text{max}} = 22$ case as compared to $E_{\text{max}} = 28$ case because more links are removed to satisfy the $E_{\text{max}} = 22$ constraint and the flow over the removed links get re-routed through shorter paths.

5. Related Work

We are not aware of any work that configures a topology to match the communication requirements of a given application.

5.1. Architecture-aware Partitioning

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., [33, 15, 34, 35, 14, 36]). 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-specific heuristic approaches. As such, generic local search heuristics have been used for solving this problem, such as genetic algorithms [37] and cross-entropy methods [38]. The few problem-specific heuristics include MinEx [39], MiniMax [40] and a variant of METIS [41].

Many heuristics for the special case of topology-aware mapping have also been proposed [9, 42] (and some software for this problem is publicly available [46]). The task of topology-aware mapping and message routing have also been studied for alternative interconnect topologies, such as a dragonfly topology [44], fat-trees [43], mesh and torus topologies [45]. In contrast to these works, our framework requires topology-aware partitioning, mapping and routing solution for a dynamic topology graph. In our framework, the underlying topology is not fixed and it can vary widely within the constraints of the reconfigurable interconnect. Furthermore, in contrast to the existing work on topology-aware mapping and routing, we focus on throughput maximization for streaming applications (by focusing directly on hot-spots created on nodes and links) rather than latency reduction based on distance estimates (average hops travelled per byte). Also, many of the existing topology-aware mapping techniques rely on domain specific knowledge about the application such as geometric coordinates associated with the physical space being simulated, while our techniques are agnostic to the internals of the streaming application.

5.2. Reconfigurable Interconnects

There is also some work on exploring reconfigurable interconnects for scientific applications. The work in [5] is very much related to ours, in the sense that it substantiates our contribution. Our contribution almost starts where the contribution of [5] finishes. Specifically, an important contribution of [5] 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.

Unlike some previous efforts (e.g., [6]) that have investigated the use of reconfigurable interconnect in high performance computing, our framework does not assume a certain class of applications or certain communication patterns.

6. Discussion

In this paper, we have proposed a framework for computing a good topology that is configurable within the constraints of a reconfigurable interconnect. Our focus has been on long-duration streaming applications, where it is worthwhile to invest some time to get a topology configuration that elicits high throughput.

For dynamic workloads, we note that the changes required in the topology and the mapping solution when the workload changes, are unlikely to be substantial as long as the dependencies between the logical functional units do not change drastically. Thus, our framework can use the existing topology and the mapping solution as the initial topology and compute just one or two iterations of the required alterations to quickly converge to the new topology.

As for the iterations themselves, we note that a proxy function has been proposed [3] to evaluate the efficacy of a topology without explicitly computing the partitioning, mapping and routing solution. This proxy function (based on Kirchhoff index of a certain weighted graph related to the interconnection network) is significantly faster to compute and can be easily parallelized. Leveraging this function in the iterations, the computationally expensive step of computing the topology aware partitioning and routing solution needs to be performed only once – at the end of the computation. Even here, the local search depth in the refinement phase (cf. Section 3.3.3) can be tuned so that it achieves the desired trade-off between running time and the solution quality. We expect that with these modifications, our framework can also be useful for computing good topologies for dynamic workloads, even when the architecture involves many hundreds of compute nodes.

References

[1] D. Ajwani, S. Ali, and J. P. Morrison, "Graph partitioning for reconfigurable topology," in *26th IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS '12)*, 2012.

[2] A. Hackett, D. Ajwani, S. Ali, S. Kirkland, and J. P. Morrison, "Seeds for a heterogeneous interconnect," in *22nd IEEE Heterogeneous Computing Workshop (HCW 2013)* in the proceedings of the 27th IEEE Int'l Parallel and Distributed Processing Symposium Workshops (IPDPSW 2013), 2013.

[3] A. Hackett, D. Ajwani, S. Ali, S. Kirkland, and J. P. Morrison, "A Network Configuration Algorithm Based on Optimization of Kirchhoff Index," in *27th IEEE International Symposium on Parallel and Distributed Processing*, pp. 407-417, 2013.

[4] 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*. Optical Society of America, 2009.

[5] S. Kamil, A. Pinar, D. Gunter, M. Lijewski, L. Oliker, and J. Shalf, "Reconfigurable hybrid interconnection for static and dynamic scientific applications," in *Proceedings of the 4th international conference on Computing frontiers*, ser. CF '07. New York, NY, USA: ACM, 2007, pp. 183–194. [Online]. Available: http://doi.acm.org/10.1145/1242531.1242559

[6] 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 *ACM/IEEE Conference on High Performance Networking and Computing (SC)*, 2005, p. 16.

[7] 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 *36th European Conference on Optical Communications (ECOC)*, 2010, pp. 19–23.

[8] T. N. Bui and C. Jones, "Finding Good Approximate Vertex and Edge Partitions is NP-hard," in *Information Processing Letters*, 42(3), 1992, pp. 153–159.

[9] T. Hoefler and M. Snir, "Generic Topology Mapping Strategies for Large-scale Parallel Architectures,", in *Proceedings of the International Conference on Supercomputing (ICS '11)*, 2011, pp. 75–84.

[10] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *Journal on Scientific Computing*, vol. 20, no. 1, pp. 359 – 392, 1999.

[11] 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*, vol. 25, no. 3, pp. 504–517, 2006.

[12] B. Hendrickson and R. Leland, "A multilevel algorithm for partitioning graphs," in *1995 ACM/IEEE Conference on Supercomputing*, 1995.

[13] C. Walshaw and M. Cross, "Parallel Optimisation Algorithms for Multilevel Mesh Partitioning," *Parallel Comput.*, vol. 26, no. 12, pp. 1635–1660, 2000.

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

[15] U. Catalyurek, E. Boman, K. Devine, D. Bozdag, R. Heaphy, and L. Riesen, "Hypergraph-based dynamic load balancing for adaptive scientific computations," in *21st IEEE International Parallel and Distributed Processing Symposium*, Mar. 2007.

[16] M. Holtgrewe, P. Sanders, and C. Schulz, "Engineering a scalable high quality graph partitioner," in *24th IEEE International Parallel and Distributed Processing Symposium*, 2010.

[17] B. Gedik, H. Andrade and K.-L. Wu and P. S. Yu and M. Doo, "SPADE: the System S declarative stream processing engine", in *Proc. of the ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, 2008, pp. 1123–1134.

[18] D. Ediger, J. Riedy, H. Meyerhenke and D. A. Bader, "Tracking Structure of Streaming Social Networks", in *IPDPS Workshops – 5th Workshop on Multithreaded Architectures and Applications (MTAAP'11)*, 2011, pp. 1691–1699.

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

[20] D. Ajwani, S. Ali, K. Katrinis, C.-H. Li, A. J. Park, J. P. Morrison and E. Schenfeld, "Generating synthetic task graphs for simulating stream computing systems," Journal of Parallel and Distributed Computing, 73(10), 2013, pp. 1362-1374.

[21] C. Walshaw. The Graph Partitioning Archive. [Online]. Available: http://staffweb.cms.gre.ac.uk/ c.walshaw/partition/

[22] The 10th DIMACS Implementation Challenge – Graph Partitioning and Clustering. [Online]. Available: http://www.cc.gatech.edu/dimacs10/

[23] S. Bujnowski, B. Dubalski, and A. Zabludowski, "Analysis of chordal rings," in *Mathematical Techniques and Problems in Telecommunications (MTPT '03)*. Centro Int'l de Mathematica, Tomar, 2003, pp. 257–279.

[24] L. Narayanan, J. Opatrny, and D. Sotteau, "All-to-all optical routing in optimal chordal rings of degree four," in *10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '99)*, 1999.

[25] J. M. Pedersen, T. M. Riaz, B. Dubalski, and O. B. Madsen, "A comparison of network planning strategies," in *10th Int'l Conf. on Advanced Communication Technology (ICACT '08)*, 2008.

[26] B. Parhami, "A class of odd-radix chordal ring networks," *CSI Journal on Computer Science and Engineering*, vol. 4, no. 2&4, pp. 1–9, 2006.

[27] ——, "Periodically regular chordal rings are preferable to double-ring networks," *J. of Interconnection Networks*, vol. 9, no. 1, pp. 99–126, 2008.

[28] N. G. de Bruijn, "A combinatorial problem," *Koninklijke Nederlandse Akademie v. Wetenschappen*, vol. 49, pp. 758–764, 1946.

[29] D. Li, X. Lu, and J. Su, "Graph-theoretic analysis of Kautz topology and DHT schemes," in *Network and Parallel Computing: IFIP Int'l Conf.*, 2004.

[30] R. F. Browne and R. M. Hodgson, "Symmetric degree-four chordal ring networks," *Computers and Digital Techniques, IEE Proc. E*, vol. 137, no. 4, 1990.

[31] Y. Iwasaki, Y. Kajiwara, K. Obokata, and Y. Igarashi, "Independent spanning trees of chordal rings," *Information Processing Letters*, vol. 69, pp. 155–160, 1999.

[32] X. Yang, D. J. Evans, and G. M. Megson, "Maximum induced subgraph of a recursive circulant," *Information Processing Letters*, vol. 95, no. 1, pp. 293–298, 2005.

[33] G. Karypis and V. Kumar, "Parallel multilevel series k-way partitioning scheme for irregular graphs," *SIAM Review*, vol. 41, no. 2, pp. 278–300, 1999.

[34] U. Catalyurek, E. Boman, K. Devine, D. Bozdag, R. Heaphy, and L. Riesen, "A repartitioning hypergraph model for dynamic load balancing." *J. Parallel Distrib. Comput.*, vol. 69, no. 8, pp. 711–724, 2009.

[35] 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 *10th ACM/IFIP/USENIX International Conference on Middleware*, 2009.

[36] A. J. Soper, C. Walshaw, and M. Cross, "A combined evolutionary search and multilevel optimisation approach to graph partitioning," *J. Global Optimization*, vol. 29, no. 2, pp. 225–241, 2004.

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

[38] S. Sanyal and S. K. Das, "Match : Mapping data-parallel tasks on a heterogeneous computing platform using the cross-entropy heuristic," in *19th IEEE International Parallel and Distributed Processing Symposium*, 2005.

[39] D. J. Harvey, S. K. Das, and R. Biswas, "Design and performance of a heterogeneous grid partitioner," *Algorithmica*, vol. 45, no. 3, pp. 509–530, 2006.

[40] S. Kumar, S. K. Das, and R. Biswas, "Graph partitioning for parallel applications in heterogeneous grid environments," in *16th IEEE International Parallel and Distributed Processing Symposium*, 2002.

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

[42] A. Bhatele, N. Jain, W. D. Gropp and L. V. Kale, "Avoiding Hot-spots on Two-level Direct Networks," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*, 2011, pp. 76:1–76:11.

[43] A. Bhatele, "Topology Aware Task Mapping," in *Encyclopedia of Parallel Computing*, Springer Verlag, 2011.

[44] N. Jain, A. Bhatele, X. Ni, N. J. Wright and L. V. Kale, "Maximizing Throughput on a Dragonfly Network," in *Proceedings of the 2014 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*, 2014, pp. 336–347.

[45] A. Bhatele, and L. V. Kale, "Heuristic-based techniques for mapping irregular communication graphs to mesh topologies," in *Proceedings of Workshop on Extreme Scale Computing APplication Enablement - Modeling and Tools*, 2011.

[46] A. Bhatele, E. Bohm, N. Jain and L. V. Kale, 2013, *http://charm.cs.illinois.edu/research/topology.*

Dr. Deepak Ajwani is a research scientist at Bell Labs Ireland. Prior to that, he was a Postdoctoral researcher at the Centre for Unified Computing, University College Cork, Cork, Ireland and at Aarhus University, Aarhus, Denmark. He received his Ph.D. from Max Planck Institute for Informatics, Saarbruecken, Germany in 2008. His research interest include algorithms for processing massive data including external memory and parallel algorithms, graph algorithms and algorithm engineering.

Dr. Adam Hackett is a posdoctoral research fellow at the Hamilton Institute, National University of Ireland, Maynooth. He received his Ph.D. in Applied Mathematics from the University of Limerick in January 2012. While there, he was a member of the Stochastic Dynamics and Complex Systems research group and an associate member of the Mathematics Applications Consortium for Science and Industry. His current research interests include application of matrix and graph theory to exascale computing.

Dr. Shoukat Ali is a Research Staff Member at IBM Research - Ireland where he is a part of the Exascale Systems Group. His current research interests include system software for large-scale computers, software-defined infrastructures (including data centers), large-scale FPGA-based emulation, energy-efficient HPC systems, graph partitioning for reconfigurable topologies, and performance analysis. He has over 40 publications in these areas. Before joining IBM Research, he worked at Intel, CA, on partitioning some of the largest graphs in the CAD industry. Before Intel, he worked at the University of Missouri-Rolla on various issues in robust resource allocation. He received his PhD and MS from Purdue University, West Lafayette, IN.

Stephen J. Kirkland is a Stokes Professor at the National University of Ireland, Maynooth. He is editor-in-chief of Linear and Multilinear Algebra and serves on the editorial boards of several other journals. His research interests are primarily in matrix theory and graph theory, with an emphasis on the interconnections between these two areas. He has over 100 journal publications in these areas.

Prof. John P. Morrison is the founder and director of the Centre for Unified Computing and a co-founder and co-Director of Boole Centre for Research in Informatics and Grid-Ireland. He is a Science Foundation of Ireland investigator award holder and has published widely in the field of parallel distributed and grid computing. He has served on dozens of international conference programme committees and is a co-founder of the International Symposium on Parallel and Distributed Computing.