# Modular design of data-parallel graph algorithms

Santanu Kumar Dash [*], Sven-Bodo Scholz[†], Bruce Christianson[*]

[*]University of Hertfordshire, Hatfield, United Kingdom

{s.dash, b.christianson}@herts.ac.uk

[†] Heriot-Watt University, Edinburgh, United Kingdom

S.Scholz@hw.ac.uk

*contact author: Santanu Kumar Dash*

# Modular design of data-parallel graph algorithms

*Abstract*—**Amorphous Data Parallelism has proven to be a suitable vehicle for implementing concurrent graph algorithms effectively on multi-core architectures. In view of the growing complexity of graph algorithms for information analysis, there is a need to facilitate modular design techniques in the context of Amorphous Data Parallelism. In this paper, we investigate what it takes to formulate algorithms possessing Amorphous Data Parallelism in a modular fashion enabling a large degree of code re-use. Using the betweenness centrality algorithm, a widely popular algorithm in the analysis of social networks, we demonstrate that a single optimisation technique can suffice to enable a modular programming style without loosing the efficiency of a tailor-made monolithic implementation.**

*Index Terms*—**multi-core, parallelisation, programming model**

## I. INTRODUCTION

With the recent advent of the internet and social networks, massive amounts of digital information is being generated today. A need to analyze this information has triggered the development of sophisticated graph algorithms [1]. The complexity of these algorithms coupled with the massive datasets that they are typically applied to creates a strong demand for an efffective utilisation of multi- and many-core systems.

Unfortunately, graph algorithms do not lend themselves well to implementation on multi-core architectures. The irregular structure of graphs makes static analysis a hard task and it is difficult to foretell the execution footprint of graph algorithms. Much of the parallelism inherent in graph algorithms comes from the complex interplay of runtime factors which cannot be modelled statically. While some of the algorithms have been successfully deployed on multi-core architectures through the use of clever heuristics [2][3], a large class of graph algorithms have eluded a parallel implementation. To ameliorate this situation, researchers have focused on identifying generic models for parallelism in graph algorithms.

One recent approach known as Tao Analysis abstracts away from the algorithmic specification and instead, formulates graph algorithms in terms of operations that are performed on the graph structure as a whole [4][5]. The key idea of the Tao Analysis is to identify a set of nodes named *active nodes* and then to collectively apply a combination of operators to all these nodes. When an operator is applied to an active node, it usually affects not only the node itself but also an area of the graph around that node. This operator-specific and potentially also node-specific area is referred to as the *neighbourhood*. The concurrency in an algorithm is exposed when operators with non-overlapping neighbourhood are executed in parallel. This form of parallelism is called Amorphous Data Parallelism (ADP) and has been shown to be prevalent in a broad class of graph algorithms [5].

ADP offers several benefits to the application programmer. Firstly, ADP paves the way for realising scalable performance on multi-core systems for a large group of graph algorithms [6]

[5] [4]. Secondly, algorithms possessing ADP properties are formulated on a single layer of abstraction that enables the encapsulation of the low-level concurrency mechanisms into highly optimised libraries. These libraries are fine-tuned to deliver efficient multi-core execution without explicit instructions from the programmer. Consequently, application programmers can focus on writing the graph algorithms without worrying about low-level concurrency tuning.

This paper investigates how far this bi-section of programming skills can be driven forward. In most published works, individual algorithms under investigation have been handcrafted with a view to performance. Re-using pre-developed codebase of other applications has been of minor concern. However, if ADP is to be used in main-stream computing it is very likely that programmers will try to adopt a programming style that attempts to maximise modularity and, with it, the amount of code re-use possible. In this paper, we investigate the potential impact such a modularised programming style may have on the overall performance achieved.

As an example, we look at the betweenness centrality algorithm - a widely used algorithm in the analysis of social networks. We implement two versions of this algorithm: a monolithic version that implements the entire algorithm within one ADP operation, and a modularised version where the algorithm is formulated as a composition of several much simpler ADP operations developed for potential re-use in varied contexts.

The main contribution of the paper is not only an extensive experimental comparison of the two versions but also an identification of an optimisation technique that allows to transform one form into the other. Furthermore, we briefly discuss the challenges in automating the optimisation within a compiler setting and highlight how expressing the algorithm in a functional language can simplify the applicability of the optimisation.

The rest of the paper is organized as follows. In section II, we give an introduction to the Tao analysis of algorithms and ADP. We give an overview of the betweenness algorithm in section III. The operator formulation of the betweenness algorithm is discussed in section IV. Section V describes the advantages of operator fusion and presents experimental results to reinforce the advantages of operator fusion. A case for implementing operator formulations in functional languages is presented in section VI. Finally, the paper is concluded in section VII.

## II. AMORPHOUS DATA PARALLELISM

The operator formulation of graph algorithms expresses algorithms in terms of actions they perform on the graph. Vertices on which these operators are applied are called active nodes. Upon application of an operation to an active node,

| Vertex ($\omega$) | Shortest paths from $\omega$ | Paths through d | $\delta_{\omega\bullet}$(d) |
|---|---|---|---|
| a | a $\rightarrow$ b <br> a $\rightarrow$ b $\rightarrow$ c <br> a $\rightarrow$ b $\rightarrow$ c $\rightarrow$ e <br> a $\rightarrow$ b $\rightarrow$ d $\rightarrow$ e | a $\rightarrow$ b $\rightarrow$ d $\rightarrow$ e | 1÷4 = 0.25 |
| b | b $\rightarrow$ c <br> b $\rightarrow$ c $\rightarrow$ e <br> b $\rightarrow$ d $\rightarrow$ e | b $\rightarrow$ d $\rightarrow$ e | 1÷3 = 0.33 |

TABLE I

EXAMPLE OF DEPENDENCE VALUE COMPUTATION FOR VERTEX D IN FIGURE 1

other nodes or edges in the vicinity of the active node may be modified. If there are no conflicting neighbourhoods for activities at two different active nodes, then those activities can be executed in parallel. Otherwise, we may need some form of locking to enable the activities to execute. The parallelism thus uncovered is known as Amorphous Data Parallelism (ADP).

While ADP seems like an intuitive concept, exploiting latent ADP in applications is a non-trivial task. Amongst other things, one needs to understand the nature of active nodes and neighbourhoods in the operator formulation. An understanding of the life-cycle of active nodes is also necessary i.e. the way in which vertices become active. It is important to comprehend whether active nodes can be executed in parallel or they need to be executed in a certain order. This largely depends on the nature of the operators that are applied to the active nodes. Therefore, it is important to study the operators themselves before deciding on a runtime scheme for scheduling and synchronizing parallel activities. In order to unravel ADP in an algorithm, therefore, we need a combination of a rigourous analytical framework as well as powerful runtime support for different scheduling and synchronization policies.

While there are standard schemes available for scheduling and synchronization, it is the initial analysis of the algorithm that is challenging and needs further elaboration. For uncovering the latent ADP in the algorithm and coming up with a suitable scheduling scheme, the analytical framework that is used is termed as Tao analysis [5]. There are three dimensions to the Tao analysis which are enumerated below.

1) *Topology*: It is important to understand the structure of the graph on which the operators are executed. This information is necessary as regular graphs are amenable to many compile time optimizations. Also, it is easier to come up with a compile time scheduling policy for algorithms on regular graphs.

2) *Active Node*: Often the execution of operators on currently active nodes spawns new ones. It is necessary to understand the manner in which active nodes come into being. This information can be used to work out the right scheduling policy while applying operators to active nodes. For the same purpose, it is important to understand whether operators on the currently active nodes can be executed in a certain order or they can be executed in any order.

3) *Operators*: A good understanding of the nature of operators is necessary in deciding what kind of locking and roll-back mechanism may be necessary for the algorithm. There are three classes of operators that have been discovered so far in the context of graph algorithms [5]. The first class of operators is the morph operator. This operator modifies the graph in the neigbourhood of the active nodes. The second class of operators are called local computations. They do not modify the graph but update the values on the vertices and edges in the vicinity of the active nodes. Finally, readers do not modify or update values on vertices and edges. Instead, they are commonly used to read these values. For most reader and local computation operators, no locking scheme is normally necessary. However, morph

operators necessitate the use of locking or roll-back schemes is essential to ensure program correctness.

### III. THE BETWEENNESS CENTRALITY MEASURE

The betweenness centrality measure is indicative of the realtive importance of a vertex in a graph. For the subsequent discussions let us assume that the algorithm operates on a graph $G = V \times E$, where $V$ is the set of vertices in the graph and $E$ is the set of edges. Betweenness centrality for a vertex $v$ is defined as the sum of the dependence of all vertices $s \in V$ on $v$ in reaching all other vertices $t \in V$.

The computation of betweenness values for a vertex $v$ is shown in equation 1. Here, $\delta_{s\bullet}(v)$ is the dependence of vertex $s$ on $v$ to reach all other vertices in the graph. Equation 2 shows how to compute the dependence of a vertex $s$ on another vertex $v$. This equation sums up the dependence of $s$ on $v$ in reaching a target vertex $t$ for all $t \in V$. Here, $\delta_{st}(v)$ is the target specific dependence of $s$ on $v$. Computation of target-specific dependence is further highlighted in equation 3. Target-specific dependence of a given source $s$ on another vertex $v$ for a given target $t$ is defined as the ratio of number of shortest paths between $s$ and $t$ passing through $v$ (denoted by $\sigma_{st}(v)$ in equation 3) to the ratio of the total number of shortest paths between $s$ and $t$ (denoted by $\sigma_{st}$ in equation 3).

$$BC(v) = \sum_{s \in V, s \neq v} \delta_{s\bullet}(v) \tag{1}$$

$$\delta_{s\bullet}(v) = \sum_{t \in V, t \neq v} \delta_{st}(v) \tag{2}$$

$$\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{3}$$

Take the case of the graph shown in figure 1 as an example. The matrix alongside the graph shows all shortest paths between pairs of vertices. Let us try to compute the betweenness value for vertex $d$ here. Only shortest paths from $a$ and $b$ to other vertices pass through $d$. Therefore, the dependence of other vertices on $d$ can be ignored in this case. Now let us look at the paths themselves. Table III shows the shortest path information from $a$ and $b$ to other vertices. It also shows shortest paths from $a$ and $b$ to other vertices passing through $d$ and the computation of dependence values from this information. Thus, the betweenness value for vertex $d$ based on the information from table III is $0.25 + 0.33 = 0.58$.

As can be observed, computation of the betweenness values involves computation of all-pair all-shortest-paths. This can
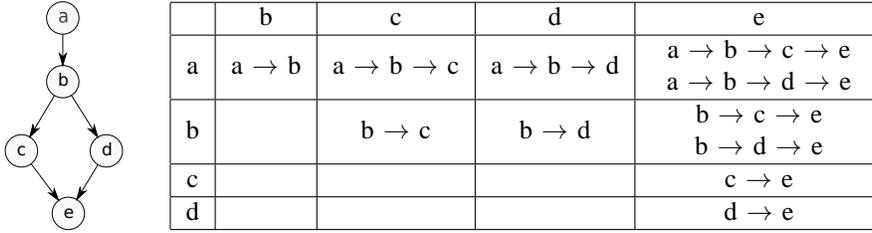
Fig. 1. All-pair all-shortest-paths for a directed graph

be very expensive even for small graphs. However, it was observed in [7] that the computation of the number of shortest paths i.e. $\sigma_{st}$ values can be done using a modified breadth first search. For the subsequent discussion, we define the depth of a node $v$ in a breadth first tree as the hop distance of $v$ from the source of the breadth first tree. The observation in [7] was based on the fact that if a parent $p$ has a depth of one less that the depth of its child $q$ in the graph for a given breadth first tree spanning the graph, then the shortest path from edge from the source to $p$ augmented with the edge from $p$ to $q$ is also a shortest path from the source to $q$.

If for every vertex $t$, we can compute a parent sub-set $P_s(t)$ such that these parents have a depth of one less than $t$ in the breadth first search originating at $s$, we can compute $\sigma_{st}$ as shown in 4. It was further shown in [7] that the $\delta_{s\bullet}$ values can be computed recursively as well using the $\sigma_{st}$ values as shown in equation 5. The details of the derivation of this expression can be found in equation 5 and the derivation uses a similar observation as the one used for simplifying the $\sigma_{st}$ computation.

$$\sigma_{st} = \sum_{r \in P_s(t)} \sigma_{sr} \qquad (4)$$

$$\delta_{s\bullet}(r) = \sum_{r \in P_s(t)} \frac{\sigma_{sr}}{\sigma_{st}}(1 + \delta_s(t)) \qquad (5)$$

There are two distinct advantages of equations 4 and 5. Firstly, they do away with the added complexity of computing the number of shortest paths passing through a given vertex $v$ (denoted by $\sigma_{st}(v)$). Secondly and more importantly, they simplify the computation of dependence values to an extend where it can be done recursively using a breadth first backtracking. In other words, if we start at the leaves of a breadth first tree and backtrack we can recursively compute the dependence values by using equation 5.

## IV. OPERATOR FORMULATION OF THE ALGORITHM

In this section, we present the Tao analysis and operator formulation of the betweenness centrality algorithm in order to uncover latent amorphous data parallelism. We discuss the nature of active nodes in the algorithm as well as the operators that operate on these active nodes. Finally, we comment upon the source of parallelism based on the operator formulation. It must be noted here that the first dimension of Tao analysis is inapplicable to the betweenness centrality algorithm as it operates on unstructured graphs. Therefore, we focus on the second and third dimensions.

### A. Active Nodes

For every vertex in the graph, we need to compute its dependence on other vertices in the graph. For a given vertex $v$, once we have the dependence of other vertices on $v$, we need to add the dependence values of all vertices on $v$ to get the betweenness centrality for $v$. This exercise needs to be repeated for every vertex. Therefore, it can be safely assumed that all vertices in the graph are active nodes in the betweenness centrality algorithm. The more important observation here is that no new active nodes are spawned during the application of operators to the currently active nodes. In the Tao analysis framework such algorithms are called topology-driven algorithms. For topology-driven algorithms, application of operators at an active node does not cause other vertices to become active. This is in contrast to data-driven algorithms where application of an operator at an active node causes other vertices to become active.

### B. Operators

As evidenced from the discussions in section III, there are four operators in the betweenness algorithm which are enumerated below.

1) *Breadth first search and registration of leaf nodes*: The first operator that is applied to the active nodes is the breadth first walk that is initiated at each active node. This walk is used to compute the level of each vertex and this is done for each of the breadth first walks. At the end of the breadth first walks, we store the leaf nodes in the breadth first tree for the backtracking and dependence value computation as described in section III.
2) *Parent sub-set and all-pair shortest path counting*: The next operator that is applied traverses the graph starting at each active node $s \in V$. The operator computes the parent sub-set $P_s(t)$ for the given active node $s$ and a given vertex $t$ such that all vertices in $P_s(t)$ are parents of $t$ and have a depth of one less that $t$ in the breadth first tree having $s$ as its root. As the vertices in $P_s(t)$ are discovered, the computation of $\sigma_{st}$ can also be completed as described in equation 4.
3) *Computation of dependence values*: A backtracking operator is then applied which starts at the leaf nodes of the breadth first trees that are recorded in the first step. During the backtracking the dependence values are recursively computed as shown in equation 5.
4) *Betweenness centrality computation*: The final operator sums up the dependence values of vertices on active

nodes for each active node. This gives the final betweenness centrality value for the active node.

Each of the operators stated above touches the entire graph. Therefore, the neighbourhood of the operators is the entire graph. One would think that this large a neighbourhood would surely impede parallel execution of the operators at active nodes. However, upon closer inspection it can be seen that for the same operator, none of the active nodes depend on results from operator applications at other active nodes. In other words, the application of operators to active nodes need not be ordered for the same operator. This observation unravels the latent amorphous data parallelism in the application. If the active nodes have their own private copies of data, then the operators can be applied in parallel without paying attention to runtime coordination.

## V. IMPROVING RUNTIME AND SCALABILITY

While the Tao analysis framework provides a nice theoretical basis for uncovering latent amorphous data parallelism in an algorithm, it does not address subtle runtime issues like memory access overheads. If the sequence of operations is not scheduled properly, it may lead to cache thrashing and the operation may ultimately become memory bound. This deteriorates algorithm runtimes and is detrimental to the scalability of the algorithm as well. In this section, we propose operator fusion as an optimization for operator formulation in order to improve runtime and scalability of the algorithm as the number of threads executing the algorithm is increased.

Consider the operator formulation of betweenness centrality as an example. There are two ways in which the operators can be applied to the active nodes in this algorithm. In the first way, each operator can be applied to all active nodes before the next operator is chosen. The other way is to apply all operators to an (or a set of) active node(s) before moving on to other active nodes. We call the former way of applying the operator *segmented operator formulation* and the later way *fused operator fomulation*. Both versions of the operator formulations for the betweenness centrality algorithm are shown in table V.

In both versions of the formulation, all vertices in the graphs are marked as active nodes and the active nodes are stored on a worklist - a data structure that holds currently active nodes. The runtime system fetches active nodes from the worklist and executes operators on them. In the segmented version of the algorithm, the four operators identified in section IV are applied one at a time to all the active nodes in the worklist. On the other hand, in the fused version, all operators are applied to each of the active nodes in one go. This ensures better cache utilization by improving data locality. As we will demonstrate shortly with experimental results, the fused version of the algorithm performs far better in terms of runtime and scalability as the number of processor threads is increased.

We executed both the segmented and fused operator formulations using two random graphs as inputs. The first graph had 8192 vertices and 60743 edges while the second graph had 16384 vertices and 123329 edges. The experiments were run on a machine with 2 Intel X5570 quad-core (giving a total of 8 cores) processors running the Linux 2.6.32 kernel. The cores shared an L3 cache of 8MB. Figure V shows the runtime and scalability measures for the two graph sizes for both the fused and segmented operators. The scalability measures are obtained by dividing the runtime for a given number of processor threads by the runtime for a single thread. The figure shows clearly how the segmented operator formulation suffers from poor runtime performance. This is attributed to ineffective cache usage. More importantly, it can be observed that the segmented version of the algorithm does not scale beyond 4 cores. Our investigations have shown that due to the massive amount of intermediate data that is generated in the segmented operator formulation, there is a significant amount of L3 cache thrashing beyond 4 threads. This makes the computation memory bound and the runtime of the algorithm does not improve despite increasing the number of threads.

## VI. A CASE FOR FUNCTIONAL PROGRAMMING

In section V, we demonstrated the importance of operator fusion in improving the runtimes of operator formulation of algorithms. While operator fusion is an essential ingredient to ensure better runtimes and scalability through better cache usage, it is not feasible to handcraft the fusion process for every composition of operators. Moreover, handcrafting the fusion process is at odds with code reuse and ease of programming where the prgrammer wishes to compose new formulations from a library of pre-developed operators. This creates a need for the fusion process to be a part of the compilation framework so that the programmer can compose formulations without being concerned about runtime overheads.

However, expressing operators in a imperative language often makes the analysis process complicated. Identification of avenues for fusion in an imperative setting is a non-trivial task given that an imperative program may contain global variables and state information which complicates data-flow analysis. On the other hand, if the operator formulation is written in a functional language, we do not have any concept of a state. Data flow analysis of a purely functional language is a much simpler task due to the referential transparency in functional languages. Consequently, the process of identifying avenues for operator fusion is greatly simplified if the operator formulation is written in a functional style. In fact, a process similar to operator fusion has already been successfully applied in the context of array programming in the Single Assignment C (SAC) language [8][9]. In SAC, this process is called with-loop folding and operations on multiple array elements are *folded* together to form a unified operation wherever possible to reduce memory overheads.

Apart from simplifying the analysis process for operator fusion, functional languages offer other benefits as well. Functional languages typically come with rich features like pattern matching, higher order functions and partial application. These features are especially useful in expressing algorithms in a succinct manner thereby boosting productivity of programmers. The power of higher order functions and pattern matching in expressing succinct graph algorithms has already been presented in [10]. With improved modularity through higher order

| Segmented operators | Fused operators |
|---|---|
| 1: G = V × E<br>2: **for all** v ∈ V **do**<br>3:     markAsActiveNode(v)<br>4: **end for**<br>5: $\mathcal{WL} \leftarrow$ activeNodes(G)<br>6: **for all** s ∈ $\mathcal{WL}$ **do**<br>7:     BFS(s)<br>8: **end for**<br>9: **for all** s ∈ $\mathcal{WL}$ **do**<br>10:     $\sigma_{calc}$(s)<br>11: **end for**<br>12: **for all** s ∈ $\mathcal{WL}$ **do**<br>13:     $\delta_{calc}$(s)<br>14: **end for**<br>15: **for all** s ∈ $\mathcal{WL}$ **do**<br>16:     $BC_{calc}$(s)<br>17: **end for** | 1: G = V × E<br>2: **for all** v ∈ V **do**<br>3:     markAsActiveNode(v)<br>4: **end for**<br>5: $\mathcal{WL} \leftarrow$ activeNodes(G)<br>6: **for all** s ∈ $\mathcal{WL}$ **do**<br>7:     BFS(s);<br>8:     $\sigma_{calc}$(s);<br>9:     $\delta_{calc}$(s);<br>10:     $BC_{calc}$(s);<br>11: **end for** |

TABLE II

SEGMENTED VS FUSED OPERATOR FORMULATION FOR THE BETWEENNESS CENTRALITY ALGORITHM
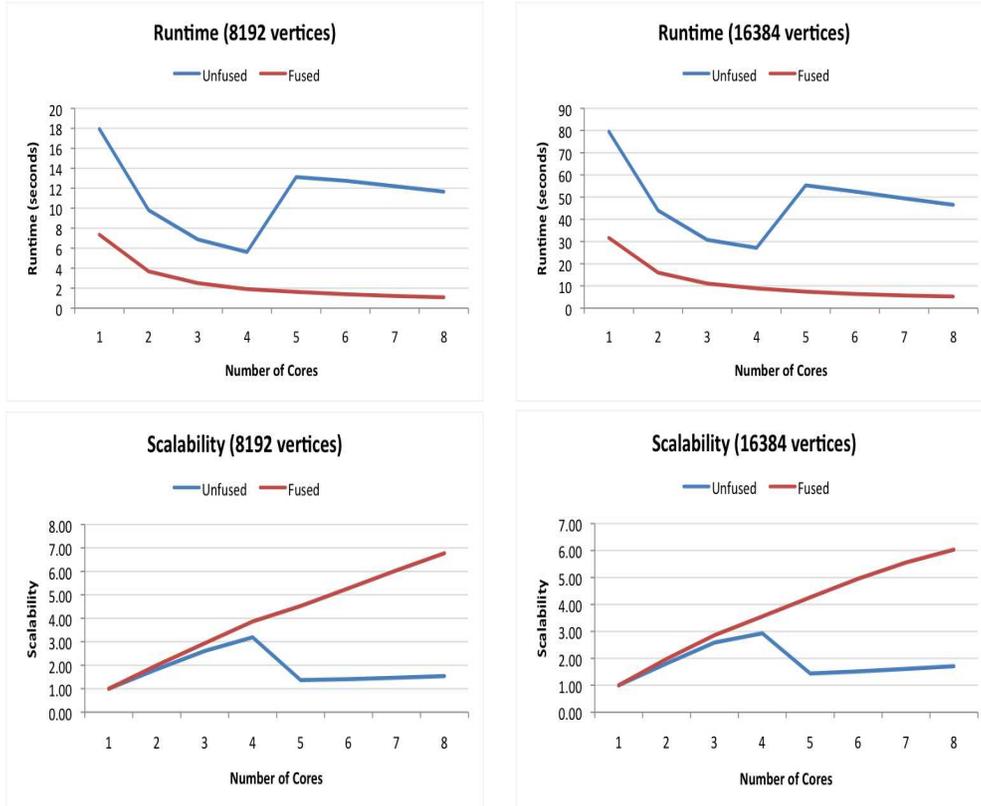


Fig. 2.    Runtime and scalability measures for random graphs of 8192 and 16384 vertices for the segmented and fused operator formulations

functions and the lack of side effects, understanding complex code segments is much easier in a functional setting than in an imperative setting. This makes maintenance of large software codebases much simpler in functional languages. With a thrust towards developing complicated techniques for information analysis using graph theory, functional languages are a good candidate for implementing complex graph algorithms.

Finally, the biggest selling point of functional languages are that they greatly simplify analysis of source code for identifying task parallelism. This is due to the Church-Rösser property of functional languages which states that in the absence of side effects, any two operations can be executed in parallel provided there is no data dependency between them. This is the reason why functional languages have been hugely successful at harnessing the power of multi-core architecture.

Due to the above reasons, we believe that extending pre-existing ideas in functional languages to the domain of amorphous data parallelism presents a compelling case for further work. It will be very interesting to explore the synergies between the two approaches and we intend to embark on this

task as a part of our future research work.

## VII. CONCLUSION

In this paper, we discussed a parallel implementation of the betweennes centrality algorithm. We did a Tao analysis of the betweenness algorithm in order to better understand the nature of active nodes and operators in the algorithm. Then, we formulated the algorithm in terms of operators that mutate information stored at active nodes. While the operator formulation of the algorithm exposed latent amorphous data parallelism in the algortihm, we showed that a naïve implementation of the operator formulation suffers from cache thrashing and memory access overheads. We showed that performance and scalability of operator formulations can be significantly improved by augmenting operator formulation with operator fusion and presented experimental results to reinforce the value of the fusion process.

While operator fusion was shown to be invaluable in optimizing operator formulations, it was discussed that analysis of source code to identify avenues for operator fusion was a non-trivial task. At the same time, it was showcased how the use of functional languages to code operator formulations eases the fusion analysis by doing away with the side effects. Given the added benefits of functional languages like code-reuse, modularity and Church-Rösser property, it can be said that functional languages are highly relevant to graph algorithms in general and parallel implementation of graph algorithms in particular. Therefore, it is worthwhile to consider the benefits of functional languages in the context of amorphous data parallelism.

## REFERENCES

[1] Ulrik Brandes and Thomas Erlebach, editors. *Network Analysis: Methodological Foundations [outcome of a Dagstuhl seminar, 13-16 April 2004]*, volume 3418 of *Lecture Notes in Computer Science*. Springer, 2005.

[2] David A. Bader and Guojing Cong. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. *J. Parallel Distrib. Comput.*, 66:1366–1378, November 2006.

[3] John R. Gilbert and Robert Schreiber. Highly parallel sparse cholesky factorization. *SIAM Journal on Scientific and Statistical Computing*, 13:1151–1172, 1992.

[4] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 211–222, New York, NY, USA, 2007. ACM.

[5] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 12–25, New York, NY, USA, 2011. ACM.

[6] Muhammad Amber Hassaan, Martin Burtscher, and Keshav Pingali. Ordered vs. unordered: a comparison of parallelism and work-efficiency in irregular algorithms. In *PPOPP*, pages 3–12, 2011.

[7] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25:163–177, 2001.

[8] S.-B. Scholz. With-loop-folding in SAC–Condensing Consecutive Array Operations. In C. Clack, K.Hammond, and T. Davie, editors, *Implementation of Functional Languages, 9th International Workshop, IFL'97, St. Andrews, Scotland, UK, September 1997, Selected Papers*, volume 1467 of *LNCS*, pages 72–92. Springer, 1998.

[9] Sven-Bodo Scholz. Single Assignment C — efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.

[10] Martin Erwig. Inductive graphs and functional graph algorithms. *Journal of Functional Programming*, 11:467–492, September 2001.