Scaling Graph Community Detection on the Tilera Many-core Architecture

Daniel Chavarría-Miranda, Mahantesh Halappanavar
High Performance Computing
Pacific Northwest National Laboratory
Richland, WA
{*daniel.chavarria, hala*}*@pnnl.gov*

Ananth Kalyanaraman
School of Electrical Engineering and Computer Science
Washington State University
Pullman, WA
ananth@eecs.wsu.edu

***Abstract*—In an era when power constraints and data movement are proving to be significant barriers for the application of high-end computing, the Tilera many-core architecture offers a low-power platform exhibiting many important characteristics of future systems, including a large number of simple cores, a sophisticated network-on-chip, and fine-grained control over memory and caching policies. While this emerging architecture has been previously studied for structured compute-intensive kernels, benchmarking the platform for data-bound, irregular applications present significant challenges that have remained unexplored. Community detection is an advanced prototypical graph-theoretic operation with applications in numerous scientific domains including life sciences, cyber security, and power systems. In this work, we explore multiple design strategies toward developing a scalable tool for community detection on the Tilera platform. Using several memory layout and work scheduling techniques we demonstrate speedups of up to 47× on 36 cores of the Tilera TileGX36 platform over the best serial implementation, and also show results that have comparable quality and performance to mainstream x86 platforms. To the best of our knowledge this is the first work addressing graph algorithms on the Tilera platform. This study demonstrates that through careful design space exploration, low-power many-core platforms like Tilera can be effectively exploited for graph algorithms that embody all the essential characteristics of an irregular application.**

***Keywords*-Tilera; community detection; many-core; parallel; graph algorithms**

I. Introduction

Graph analytics find pervasive application in numerous scientific disciplines. The inherent inter-connectedness of experimentally procured data lends itself to graph-based representations and models. Enabling advanced analytics on these graphs is key to gaining a fundamental understanding of the processes that govern naturally-built or human-built system. Community detection is an advanced graph operation capable of providing such fundamental insights. It has the potential to reveal how a given network is organized into structural modules of entities (known as "communities") that are likely to be involved in interplay within a system. Given an input graph $G(V, E)$ (weighted or unweighted), the goal of community detection is to partition the set of vertices into an arbitrary number of groups (or communities) such that the vertices within each community are highly correlated to one another and sparsely correlated to the outside world. The problem is different from the classical graph partitioning in that neither the number of communities nor their size distribution is known *a priori*. Community detection has been shown to be computationally intractable [1], and consequently, efficient heuristics are sought after in practice. Section VIII presents a brief literature review on the topic.

Despite the presence of efficient heuristics, community detection continues to be a computationally daunting task in practice. As the sizes of real world networks continue to grow into graphs containing billions of edges and beyond, the limits of current methods are put to test. On the other hand, many cutting edge high-performance systems use variants of many-core processors to achieve high-performance within tight power envelopes. Under this setting, efficient mapping of advanced graph analytic kernels such as community detection to many-core systems may hold the key to achieving the high levels of performance and throughput required to analyze massive graphs and networks at scale. The principal challenge behind graph analytics is, however, the prevalence of highly irregular memory access patterns and dynamic levels of parallelism that do not match the regular structures and memory hierarchy policies of the underlying target architectures.

The TileGX36 many-core processor is an emerging, low power architecture that provides 36 VLIW 64-bit cores, each executing up to three instructions per cycle. The tiles are connected in a 6×6 mesh by several high-performance network-on-chip (NOC) fabrics, including a network for shared memory coherence traffic and a separate network for user-programmable message-passing traffic. The system provides a coherent, shared memory address space for all the cores. The Tilera processor also provides fine-grained control over memory allocation and affinity to the cores, allowing for application-specific mapping of memory layouts to optimize locality and reduce traffic over the NOC.

Most of the application exercises on the Tilera platform have so far focused on porting structured algorithms and/or compute-intensive kernels such as biological sequence alignment [2] or FFT [3]. To the best of our knowledge, data-bound compute applications with irregular computational footprints such as graph kernels have not been studied.

A. Contributions

In this paper, we conduct a detailed study on how to efficiently map a widely used community detection heuristic called the *Louvain* method [4] to the Tilera TileGX many-core system. We choose the Louvain heuristic because of its wide adoption[1] and its demonstrated ability to produce high quality outputs [5]. Our contributions are as follows:

i) We present various design strategies aimed at improving performance of community detection on the TileGX system by utilizing application-specific locality-aware data structures and memory layouts that exploit its fine-grained locality control.

ii) We explore two different adaptive parallelization techniques for the principal kernel in the application: a load balance centric, locality-oblivious strategy; and a locality-first strategy with load balance considerations.

iii) We present a technique based on distance-1 graph coloring that facilitates fast convergence of the algorithm while at the same time providing adequate degree of parallelism at each step.

iv) Our experimental results demonstrate speedups of up to **47**× on 36 cores of the Tilera TileGX36 platform over the best serial implementation, and also results that have comparable quality and performance to mainstream x86 platforms.

II. An overview of the Tilera TileGX36

The Tilera TileGX36 system implements a many-core processor based on a two-dimensional mesh topology. Each core (called a "tile" in Tilera's terminology), consists of a 3-way VLIW processing unit, a private 32KB, 2-way set associative L1 data cache, a private 32KB, direct-mapped instruction cache and a 256KB, 8-way set associative unified L2 cache. The cache line granularity is 64 bytes across all three caches. The Tilera processor supports multiple page sizes from 64KB (default) to 16 MB (huge). Each tile is connected via multiple network routers to several networks in a mesh configuration, including a network for coherence traffic, an user-programmable message passing network, and a dedicated I/O network.

Tilera's caching policies are the salient features that we exploit in this work. For each individual memory page, the system can set the *home* tile of its data in the cache subsystem. This means that the *home* tile keeps the master copy of the data for that page in its L2 cache. Other tiles that write to that page must send their coherence updates over the NOC to the home tile in order for their updates to be realized. There are two principal modes for setting the home tile of a memory page:

- **Homed page**: A particular tile can be the home tile for the *whole* page.
- **Hashed page**: Individual cache lines in a page (64 bytes) are distributed in a round-robin manner to the L2 caches of all the tiles.

The default memory allocation policy hashes all pages except the pages that correspond to the program's stack (`allbutstack`).

Tilera provides an interface to these platform-specific memory policies and other elements via the Tilera Multicore Components (TMC) library. In particular, the library enables the allocation of memory using custom allocation policies at the granularity of memory pages (`tmc_alloc`). It also provides a heap manager interface (`tmc_mspace`) that use custom allocation policies. We take advantage of these two interfaces to create custom memory layouts for our target application. The TMC library also provides interfaces for synchronization, setting thread affinity to specific hardware tiles, as well as for using the user-programmable message passing network.

III. The Louvain algorithm and its parallelization on Tilera

The Louvain method [4] for community detection is a fast heuristic for determining community structures within a graph. Given a weighted graph $G(V, E)$, where V and E denote the sets of vertices and edges, the heuristic aims to partition V into an arbitrary number of possibly variable-sized "communities". Internally, the method uses an objective function called modularity [6], which is a measure of the quality of partitioning — intuitively, the idea is to detect communities of vertices such that the members of the same community are highly correlated to one another and sparsely correlated to the outside world. Modularity is a numerical score between 0 and 1, and the closer it is to 1, the more structurally organized the network is.

The Louvain algorithm for community detection is a multi-phase, multi-iteration algorithm capable of producing a hierarchy of communities. At the start of each phase, each vertex is assigned into a community of its own. Within each iteration, the set of vertices is linearly scanned (in an arbitrary but predefined order) and a greedy decision is made on which of the neighboring communities should that vertex migrate to in order to maximize the modularity gain. This calculation can be performed in $O(1)$ time per neighboring community assuming appropriate data structures can be maintained. Alternatively, a vertex may choose to stay in the same community if none of the migrations could provide a positive modularity gain. The iterations are continued until there is no more appreciable modularity gain (defined by a threshold) achieved between successive iterations. At this point, the current phase terminates, and a transformed graph is constructed by collapsing each community reported as of the last iteration of that phase into a meta-vertex and accordingly introducing self-edges and edges between those meta-vertices to reflect the strengths of intra-community and inter-community connections, respectively. This transformed

[1] As of this writing, the original paper has well over 1,700 citations.

graph becomes the input to the next phase and the entire process is repeated until no more appreciable modularity gain is obtained. The algorithm has $O(m+n)$ time and space complexities per iteration, where $n = |V|$ and $m = |E|$.

A. Challenges

There are multiple challenges in parallelizing the Louvain algorithm, the foremost being the sequential nature in which the vertices are visited within each iteration and the impact it has on convergence. Visiting the vertices sequentially gives the advantage of working with the latest information available from all the preceding vertices. Furthermore, updating community structures in parallel could potentially introduce the risk of negative modularity gains that could further delay convergence. In a recent work [7], we proposed multiple graph heuristics to effectively break the sequential barriers imposed by the Louvain method and speedup community detection without compromising on quality. This parallel Louvain approach, which we refer to as *Grappolo*[2], was implemented using OpenMP for standard multicore platforms and our most recent implementation delivers speedups of up to 16× on a 32-core Intel multicore machine. In the interest of space, we refer a reader interested in knowing the details of the parallel algorithm to the original paper [7]; instead, in what follows, we focus on the key challenges that we faced while porting the multi-core implementation to the Tilera many-core architecture, our proposed solutions and results.

Porting Grappolo to the Tilera many-core platform presents several new design challenges. One of the key heuristics used in Grappolo to ensure fast convergence on the multi-core platform is distance-1 coloring of the graph. The main idea was to avoid making concurrent decisions of neighboring vertices, by processing only those vertices of the same color in parallel. This scheme, while demonstrated to be effective on multi-core platforms, has the potential drawback of offering reduced parallelism while processing color sets with small number of vertices. This is particularly important for many-core architectures such as Tilera, which has the capacity to host significantly more cores (currently, up to 72 cores) than traditional multi-core models. Note that the size distribution of color sets is a strict function of the input graph's topology.

Secondly, the skewed degree distributions expected out of real world inputs alongside a need to maintain various auxiliary data structures (for keeping track of communities and modularity statistics) pose several challenges related to data locality and load balancing. These challenges are only exacerbated by the memory hierarchy resident on the chip including the need to adapt irregular access patterns and data sizes to fixed width cache lines and page sizes - as described in Section II.

Thirdly, the lightweight nature of the individual cores (limited by power envelope) and the limited L2 cache available locally on each core presents another layer of challenge compared to the standard Intel x86 or AMD architectures in multi-core systems.

IV. Optimizing graph coloring

Coloring is an effective way to reduce the number of iterations required for convergence [7]. Even though the cost of performing coloring can be made negligible (1-2% of total time), as was demonstrated in Grappolo through the use of a parallel coloring implementation [8], the size distribution of color sets produced could have a significance bearing on the overall performance of the individual iterative steps of the Louvain heuristic, as was described in Section III-A. Therefore, careful attention needs to be paid on how colors are used and assigned to vertices.

The default implementation in [8] attempts to minimizes the total number of colors used by always choosing the minimum color (label) available for a given vertex. As a consequence, there exists a skew in color set size distributions, with color sets with lower indices accumulating large number of vertices while the higher index color sets are small. The presence of numerous small color sets could reduce the degree of parallelism available during the main iterative steps of the algorithm.

In this paper, we present a heuristic of *bundling colors* to overcome the above challenge. More specifically, we begin with a sorted vector of color indices based on the color set sizes. We scan through the vector in a non-ascending order of the number of vertices per color, and stop when the total count of vertices considered is at or above a user-specified $\tau\%$ of the total number of vertices; for this paper, we use 20% in our experiments, based on the $80-20$ rule that we observed with the color set distributions of many inputs that we tested. The colors considered up to this point are processed in an independent manner – one color at a time. However, all the vertices belonging to the remaining colors are bundled together and are processed concurrently.

The tradeoff involved in bundling the remaining 80% of vertices into one large set for concurrent processing is that it could potentially allow for neighboring vertex decisions made in parallel and that could delay convergence. However, given the small color set sizes in this group, the negative impact of this scheme on convergence is also expected to be proportionately minimal. In addition, the remaining 80% of the vertices are expected to benefit from access to the updated community information available from the top 20% of vertices. Bundling of small color sets allows for better load balancing on the Tilera platform and differentiates this implementation with Grappolo. The same technique is also used in our experiments on the x86 platform for consistency. These expectations are empirically corroborated in our observations in Section VI.

V. Memory Layout and Parallelization Schemes on Tilera

For implementing graph community detection on Tilera, we used OpenMP and platform-specific extensions for mem-

[2] Italian word for a cluster (of grapes).

ory layout control and private heap management, thread synchronization and atomic operations, as well as pinning logical OpenMP software threads to specific hardware cores on the chip — as described below.

A. Memory Layout

The primary data structures used in Grappolo are arrays and hash tables. The graph data is represented using a Compressed Sparse Row (CSR) format with two arrays: a vertex pointer array and an edge array. The vertex pointer array indicates the beginning position of the list of edges for each vertex in the edge array. Each element in the edge array stores the destination vertex and weight for the edge (the source vertex is implicit). Other arrays are used to store the community membership and the degree of each vertex. Hash tables are used to build transformed graphs required for each phase.

Most of our data layout optimizations focus on the CSR arrays. The **local** and **hashed** layouts are used for the auxiliary arrays as well as the hash tables. Table I summarizes the different data layouts explored in our design.

The **partitioned** data layout is used in order to determine a potential lower bound on performance by maximizing locality and possibly reducing inter-tile coherence traffic by allowing each tile to focus on its "owned" subgraph. However, in practice, the static partitioners are fairly slow and exhibit an increasing cost for larger numbers of partitions, in addition to requiring renumbering and reordering the vertices and edges. We believe our heuristic data layouts to be of more practical value for large-scale real world inputs.

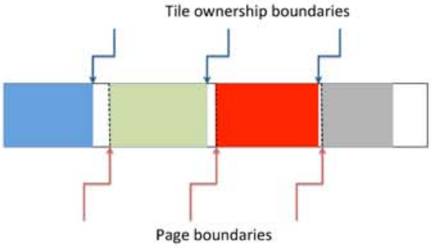


Figure 1. Padded data layout for 4 tiles

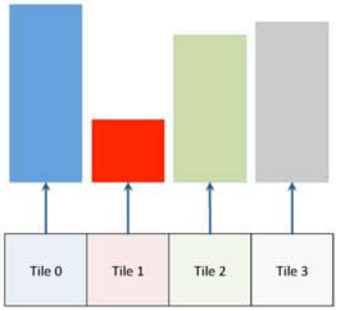


Figure 2. Partitioned data layout for 4 tiles

1) Design and Implementation: We implemented the Tilera's version of the community detection application in C++, also taking advantage of the flexibility of the data structures in C++'s Standard Template Library (STL) to use custom memory allocators [9]. We designed two Tilera-specific memory allocators that are plug-and-play compatible with the standard C++ allocator:

- **`local_allocator`**: We allocate a `tmc_mspace` heap manager for each tile running the application. We set the allocation policy to use huge pages (16 MB) homed on the particular tile. By using a heap manager, the amount of local memory allocated to each tile can grow dynamically as the application runs.
- **`segment_allocator`**: We use the `tmc_alloc` interface to allocate a contiguous block of memory rounded up to a multiple of the small page size (64 KB) — i.e., if n bytes are needed, where n is not necessarily a multiple of the page size, we allocate $\left\lceil \frac{\lceil n/p \rceil}{z} \right\rceil \times z \times p$ bytes, where z is the small page size and p is the number of tiles running the application. The segment allocator allocates at least a page for each tile.

We designed a standard interface for the CSR-based graph data structure that hides the underlying memory layout of the arrays from the calling code: the code specifies which vertex or edge to get information about and does not need to be concerned about the specific layout of the data. This flexibility enables us to experiment and determine the most efficient and best performing layouts for the graph's data structures.

The **padded** layout shown in Figure 1 uses the segment allocator to create enough data space to hold the graph's vertex and edge lists. However, each tile's block of vertices or edges starts at an exact 64 KB page boundary. The total array allocation is padded with empty space to match a multiple of the page size. The graph interface hides this padding by skipping the empty elements on the boundaries between tile blocks as necessary. The CSR vertex and edge lists are still allocated as contiguous blocks of memory in this layout. This graph layout will have some constant overhead per vertex or edge access, as it requires an integer division, integer modulo and integer multiplication operations to compute the position of a vertex or edge in the padded layout.

The **partitioned** layout show in Figure 2 does away with the contiguous allocation of the padded layout. Each tile allocates in its local memory a section of the vertex and edge lists (using the `local_allocator`). These memory blocks are separate and not contiguous. However, the external interface to the partitioned graph still maintains the same semantics as the contiguous CSR representation. The partitioned layout was designed to allow arbitrary distributions of vertices and edges onto the cores. For this reason, a small lookup table is required to determine on which local

Table I
MEMORY LAYOUT SCHEMES USED IN THIS WORK

Scheme	Description	Comments
hashed	Memory distributed in round-robin blocks of 64 bytes across tiles	Limited data locality
local	Local home pages used for private per-thread data, hashed for everything else	Exploit locality where it's clearly available
padded	Principal arrays partitioned into $\frac{n}{p}$ chunks, private data is local	Chunks are rounded up to 64KB page size, memory accesses are explicitly aligned to match page boundaries
partitioned	Vertex and edge lists are partitioned according to external partitioner input	Use PaToH and METIS as partitioners for initial graphs, local memory is used for each partition

allocation a particular vertex or edge resides. The lookup table is implemented as a linear array of ranges: element i of the range indicates the contiguous block of vertices or edges that tile i owns (v_i, v_{i+k}). The array is kept sorted in increasing range order and a binary search is performed for each vertex or edge access. However, given the fact that this array contains only up to 36 entries the lookup time for the binary search is bounded and small compared to the number of vertices or edges in the graph.

B. Graph Partitioners

In order to quantify the impact of enhanced locality via our custom data layouts, we experimented using them for our original input graphs as well as for graphs that have been partitioned using well-known, high quality static partitioners. Our motivation for using the partitioners is *only* to explore the potential benefit relative to blocked or hashed schemes for data distribution. We do not recommend the use of graph partitioners in order to execute our algorithm. In fact, we demonstrate in Section VI that there is *no* particular advantage in using partitioners.

We use the static partitioners METIS [10] and PaToH [11] to partition the input graphs in an offline process. We generate specific partitioned instances of the graphs for each experimental configuration from 2 to 36 cores. Vertices are assigned to specific partitions and edges are assigned to the partition where their source vertex originates from.

For METIS, we provide the input graph and specify the number of desired partitions using the multilevel k-way partitioning routine `METIS_PartGraphKway`. The resulting partitions are then used to relabel the graphs into contiguous subgraphs representing each partition. Each core "owns" a subgraph and is responsible for processing all the vertices belonging to its subgraph. METIS optimizes to produce load balanced partitions that minimize the total number of cross-edges. A cross-edge has its two endpoints owned by different partitions.

For PaToH we model each input graph as a hypergraph and use a multi-constraint setup that attempts to minimize the number of cross-edges while maintaining a balanced number of vertices and a balanced sum of degrees of the vertices in each partition.

C. Parallelization Schemes

As described in Section III, a major portion of the execution time of the application is spent executing Grappolo's iterations. The cost of processing a vertex within an iteration depends on its degree and the community assignments of its neighbors. For this reason, a static assignment of vertices to each parallel thread is likely to result in dynamic load imbalance during execution. Our basic parallelization strategy for each iteration uses OpenMP's `guided` scheduling mechanism to address the potential for dynamic load imbalanced across threads. `guided` scheduling will assign chunks of vertices to each thread, keeping in reserve a number of vertices for later assignment. When a thread finishes its initial chunk of vertices, it receives a smaller chunk from the scheduler. As threads deplete the reserve vertices, each unit of scheduling (chunk size) becomes smaller thus smoothing out the load imbalance while paying a modest cost in terms of scheduling overhead.

While `guided` scheduling is effective in reducing load imbalance, it does not exploit locality information related to the memory layout of the graph. Vertices are assigned to threads according to load balance concerns rather than ownership of the data by the executing core. In order to address this challenge and quantify the impact of exploiting locality in the graph structure, we designed a task-based execution strategy that executes *local* tasks first before load balancing across threads using a simple work stealing mechanism[3].

The task-based scheme works in conjunction with the initial iteration of a phase, and uses two sets of queues: one for colored vertices and one for batched vertices. Vertex processing in this initial iteration happens in two steps:

1) process vertices that belong to "large" colors in parallel; synchronizing between colors, and
2) process vertices that do not belong to "large" colors in a single batch.

The core that "owns" a vertex in a given data layout is responsible for making the modularity-based decision for migration. Due to the skewed degree distribution typical of real world networks, locality-based processing may not

[3] Our initial intent was to use OpenMP 3.0's `task` mechanism, however the GCC compiler and tasking runtime are not yet well tuned for Tilera's many-core architecture and topology, resulting in uneven performance and poor load balance.

be adequate for guaranteeing performance in this scheme. After a core has processed its portion of colored vertices, it will scan the overall set of vertices of the same color and it will attempt to atomically "steal" the execution of that vertex by marking a `processed` array of integers with its thread ID in the corresponding position for the vertex. Before a thread processes any vertex it checks to see if the `processed` array entry for that vertex has already been marked by another thread, if it is already marked then the vertex is skipped. All threads stop looking for vertices to process once all the vertices of the same color are marked as processed by a thread. An equivalent scheme is used to process batched vertices. Figure 3 illustrates this scheme in which each core processes vertices it owns and then cores with less work will go and steal "remote" vertices from other tiles. The figure also shows the state of the processed array after each core has processed two vertices.

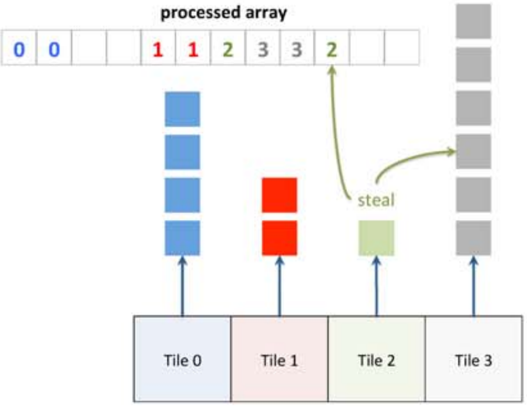


Figure 3. Task-based work stealing scheduling for locality and load balance

The intent of this task-based scheme is to try and increase the likelihood of vertices with data belonging to a particular core, being processed on that same core, while still performing load balancing as needed. Given the extra setup and preprocessing cost for this task-based scheme (determining ownership of vertices, scheduling tasks on queues), it is only used for the first Louvain iteration, where the expectation is to amortize this cost over the execution due to improved locality.

VI. Experimental Results

We present results from running a set of inputs with different characteristics that are a subset of the inputs used by Lu *et al.* [7]. We provide results from running the algorithms on the architectures described below, and from also comparing our parallel implementation with the serial implementation of Blondel *et al.* [4] in terms of result quality and improved performance.

A. Experimental setup

Experimental Platforms: We used two platforms for our experiments: the Tilera TileGX36 presented in Section II and an Intel Xeon X7560 platform. The TileGX36 platform is equipped with 32 GB of DDR3 memory separated into two 16 GB banks, with the cores running at 1.2 GHz. It runs a custom version of Linux adapted for Tilera's hardware. The compiler and runtime environment are adapted from GCC 4.7.3 and retargeted for the TileGX's 64-bit VLIW cores. The community detection code has been parallelized using OpenMP and Tilera-specific extensions for memory management, synchronization and atomic operations. OpenMP threads are pinned to different hardware cores on the Tilera chip in increasing thread ID order.

The Intel multicore platform has four sockets and 256 GB of memory. Each socket is equipped with eight cores running at 2.266 GHz, leading to a total of 32 cores. The system is equipped with 32 KB of L1 and 256 KB L2 caches per core, and 24 MB of L3 cache per socket. Each socket has 64 GB of DDR3 memory with a peak bandwidth of 34.1 GB per second. We use GCC 4.8.2 to compile a version of the code that uses OpenMP for its parallelization with the Tilera-specific extensions removed. We use `numactl` to pin OpenMP threads to cores on this platform.

Dataset: The dataset for experiments comprises of five real-world input graphs that are summarized in Table II[4]. With the exception of "MG2", all other inputs were downloaded from the DIMACS10 challenge website [12], and the University of Florida sparse matrix collection [13]. MG2 is constructed from ocean metagenomics data.

Table II
Input statistics for the real world networks used in our experimental study.

Input graph	Num. vertices (n)	Num. edges (M)	Degree statistics (λ)		
			max.	avg.	RSD
CNR	325,557	2,738,970	18,236	17	13.02
Channel	4,802,000	42,681,372	18	18	0.06
Europe	50,912,018	54,054,660	13	2	0.23
uk-2002	18,520,486	261,787,258	194,955	28	5.12
MG2	11,005,829	674,142,381	5,466	123	2.37

Code Variants: We created several versions of the code that utilize the different memory layout and work scheduling variants presented in Section V. Table III summarizes the variants used for our experiments.

An orthogonal axis for the code variants is the use of `guided` scheduling or task-stealing for the initial set of iterations under coloring. We discuss the performance and trade-offs of all these variants.

We were able to run all the code variants for the CNR, Channel, Europe and UK-2002 data sets. However, the memory requirements for the MG2 data set did not allow us to run the PaToH partitioner on it. We were able to run the METIS partitioner on MG2 and execute the *padded-metis* variants; however we also ran into resource limits on

[4] "RSD" stands for the relative standard deviation of the graph's node (unweighted) degrees, and is given by the ratio between the standard deviation of the degree and the mean.

Table III
CODE VARIANTS FOR THE COMMUNITY DETECTION APPLICATION

Layout	Details
hash	Default memory layout used hashed pages
local	Use homed pages for thread-private data, shared data is hashed
padded	*local* + use padded layout for shared data
patoh/metis	*local* + use partitioned layout for shared data; reverts to even distribution of vertices and edges per core after first phase
padded-patoh/metis	*local* + use padded layout for shared data; reverts to uniform padded layout after first phase

the local heaps managed by the **`local_allocator`** and could not run the plain *metis* variant.

B. Comparison with serial algorithm

We summarize the performance and quality of output of parallel implementations on Tilera and Intel multicore architectures compared with the serial implementation of Blondel *et al.* in Table IV.

As can be seen in Table IV the quality of the results in terms of the modularity score is consistently better for our parallel version on both platforms. Additionally, the performance of the parallel versions shows significant speedup compared to the sequential version on both platforms. The performance of the Tilera platform is highly competitive with the Intel platform in spite of its slower in-order cores, mainly due to improved parallel scalability of the code on Tilera (better utilization of the 36 cores). The exceptions to this case are for the Europe and MG2 input sets. More details are discussed in Section VII.

C. Strong scaling

We now present strong scaling results of the parallel algorithm on Tilera for three selected input sets (Channel, Europe and UK-2002) in Figure 4. We omit results for CNR and MG2 since CNR is fairly small and we ran into memory limitations to run all variants of MG2. Since we use the distance-1 coloring heuristic in all our experiments, we present the results using a single core of Tilera as well. Note that the use of coloring speeds up the computation by requiring fewer iterations. From the results provided, we observe excellent speedups for different schemes.

The speedups provided by the parallel version are significant for all inputs in comparison to the serial implementation. In many cases, the speedups are super-linear due to improved algorithmic factors in the parallel code (use of graph coloring for the initial Louvain iteration). We observe that for most cases the best performing versions are the combinations of *local* layout together with `guided` scheduling. The speedups starting at one core are above 1.0 for UK-2002, while the one core execution of Channel is slower than the serial Blondel *et al.* version (speedup of 0.31), we make up for it at scale achieving a speedup over 16.0 on 36 cores.

D. Relative performance of different layout schemes

We now provide results on relative performance of different schemes using performance profile plots in Figure 5. The Y-axis in these plots represent the fraction of inputs. The X-axis represent the factor by which a given scheme compares to the best performing scheme for a given fraction of inputs. Intuitively, the closer the lines are to the Y-axis, the better they consistently outperform the other schemes.

We can observe from the plots that the performance of the *local* guided variant is the best overall, while the default *hash* data layout is consistently worse than others. We also observe that the overhead required for more complex schemes in terms of memory layout and work scheduling has an adverse impact leading to a better performance of the much simpler *local* guided scheme for a large fraction of inputs.

VII. ANALYSIS AND DISCUSSION

The results presented in Section VI demonstrate that the performance of Tilera platform for community detection is highly competitive with the Intel Xeon platform in spite of the relatively lower performance of Tilera's simple in-order VLIW cores compared to the heavyweight x86, out-of-order cores. The improved scalability on Tilera leads to an overall better performance at the full-system scale.

An analogous situation appears with respect to different variants of the code that we tested. The simple *local* data layout combined with the *guided* scheduling policy to improve load balance outperforms the other complex layout and work scheduling schemes that require higher setup and execution costs to improve locality. The use of hashed pages for the shared data structures together with homed pages for the purely private per-thread data results in a winning combination for most input cases. The reason for this stems from the low spatial and temporal reuse of data in this application. Each vertex in a graph is visited once per iteration with a large reuse distance until the next iteration.

The single instance where the Tilera system was slower than the x86 system was for the Europe input set. This graph is a highly connected input and in contrast to other inputs, the first phase does not result in a significant reduction in the size of the graph for the next phase. The reduction in size is only a factor of $2\times$ compared to a factor of $11\times$ for Channel and $8\times$ for MG2. This small reduction results in a relatively large graphs in subsequent phases that are also relatively well connected. We use a set of private per-thread hash tables to keep track of the neighboring communities of a vertex. In the case of Europe, the number of neighboring communities per vertex remains substantially high for graphs in later phases, and therefore lead to larger memory management costs for hash tables. This overhead is usually negligible for other graphs that are characterized by a quick reduction in size for advancing phases. The lower sequential performance of Tilera cores exacerbates this problem compared to the x86 platform.

Table IV
RELATIVE PERFORMANCE OF THE SERIAL IMPLEMENTATION OF BLONDEL *et al.* AND OUR PARALLEL CODE ON TILERA AND INTEL MULTICORE PLATFORMS FOR FULL SYSTEM CONFIGURATIONS (36 AND 32 CORES RESPECTIVELY). THE NUMBERS REPORTED FOR TILERA CORRESPOND TO THE PERFORMANCE OF THE BEST VERSION ("LOCAL-GUIDED", EXCEPT FOR CNR (PATOH) AND EUROPE (METIS)).

Input	Serial (Tilera)		Parallel (Tilera/36)		Serial (Intel)	Parallel (Intel/32)	
	Modularity	Time(s)	Modularity	Time(s)	Time(s)	Modularity	Time(s)
CNR	0.912784	37.89	0.912497	**0.87**	4.36	**0.912626**	1.25
Channel	0.849672	287.65	0.934461	**17.57**	30.92	**0.934671**	25.58
Europe	–	–	0.998843	335.00	–	**0.998846**	**163.03**
uk-2002	0.9897	2,340.16	0.989526	**50.20**	335.99	**0.989532**	52.18
MG2	0.998426	4,011.61	0.998416	159.63	1313.74	**0.998426**	**101.96**

VIII. RELATED WORK

Community detection is one of the more advanced graph operations that finds pervasive application in a number of scientific domains [5]. In their seminal work, Newman and Girvan [6] introduced the modularity metric to assess the quality of a given partitioning of vertices into communities. Based on this metric as the objective, a number of heuristics have been proposed for community detection. These heuristics can be broadly classified into two categories — divisive and agglomerative. Divisive approaches [6], [14] use a top-down approach, by breaking the graph iteratively into smaller partitions using measures such as betweenness centrality. These methods are generally slower ($O(n^3)$ for sparse inputs). Agglomerative methods [15], [16] use a bottom-up approach in which larger communities are formed by greedily merging existing smaller communities. The Louvain method [4] can be viewed as a variant of the agglomerative strategy, where merging decisions are made at the individual vertex level rather than communities.

Efforts to parallelize community detection heuristics have been more recent. Riedy *et al.* present a highly parallel multithreaded implementation of the Clauset-Newman algorithm [17]. Auer and Bisseling [18] present a graph coarsening based approach to perform agglomerative clustering using GPUs. More recently, multiple parallel implementations of the Louvain heuristic have been developed [7], [19], [20]. All of these efforts have been on traditional multicore platforms. To the best of our knowledge, there has been no prior work in studying community detection on the Tilera many core architecture.

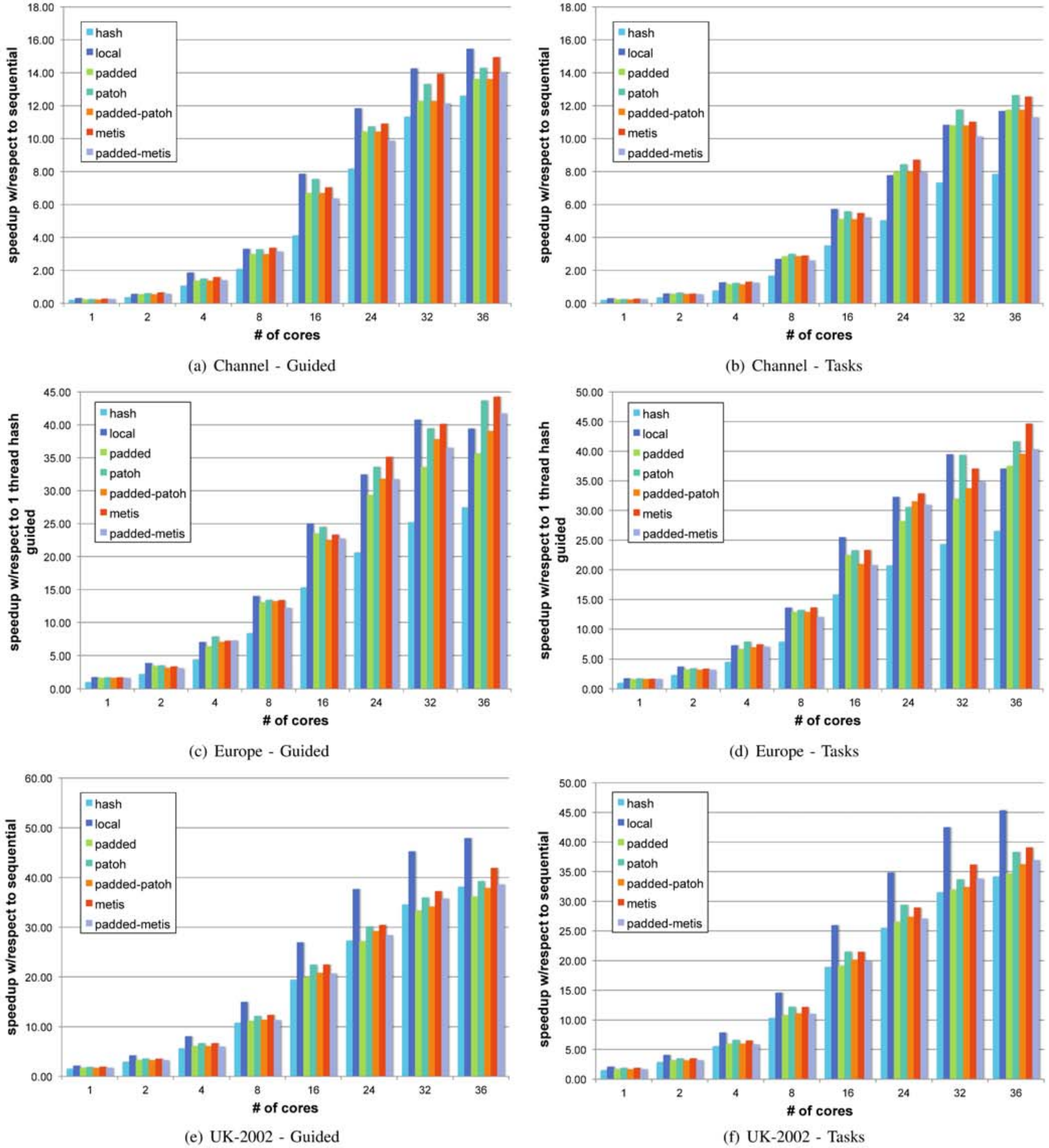
Related work on Tilera: Jagtap *et al.* [21] discuss their experience on mapping and optimizing a discrete event simulation framework to the Tilera TilePro64 system. Communication and interaction patterns between parallel threads on discrete event simulations resemble the challenges we found on our irregular community detection application. They also focus on dynamic load balance and found that object placement on the cores did not have a large impact on performance. Galvez *et al.* [2] discuss their experience on mapping the NeedlemanWunsch sequence alignment algorithm to the Tilera TilePro64 platform. This problem involves computing a highly structured table. The authors report speedups of over 20× compared to an optimized sequential implementation of the algorithm. Safari *et al.* [22] develop stereo vision algorithms for the Tilera platform, for application in power constrained mobile, space-based sensing. Their results show improved performance compared to other platforms, except for ASIC-based hardware implementations. Hung *et al.* [23] discuss a related application: object detection in video analysis. They also demonstrate good performance and thus suitability for embedded applications. Morari *et al.* [24] presented an optimized implementation of radix sort on the Tilera TilePro64 system which is another data-intensive application with somewhat less irregularity than community detection. They also observed a major impact on performance of the application with respect to the cache homing policy. Berezecki *et al.* [25] present a performance and power study of another irregular application, that of storing key-value data structures in the context of data center applications, on the Tilera many-core architecture.

IX. CONCLUSIONS

We presented a detailed set of memory layout and scheduling techniques for efficient implementation of a popular community detection algorithm on the Tilera many-core architecture. Supported by experimental results we demonstrated not only excellent scaling on the Tilera platform, but also competitive performance relative to traditional x86 multi-core architectures. Using algorithmic enhancements and efficient implementations we show a speedup of 47× on 36 cores of TileGX36 relative to the best serial implementation. We further demonstrate that the overhead costs involved with graph partitioners and sophisticated work scheduling adversely impact performance, and simpler methods that exploit data locality and simple load balancing perform better. As power limitations impose restrictions on emerging architectures and drive toward systems with a larger number of weaker cores, the presented work on a prototypical irregular application demonstrates promise of better performance on future low-power architectures and guide other researchers on developing their applications to target such architectures.

ACKNOWLEDGMENT

The research was in part supported by DOE award DE-SC-0006516, and the Center for Adaptive Super Computing Software Multithreaded Architectures (CASS-MT) at



(a) Channel - Guided

(b) Channel - Tasks

(c) Europe - Guided

(d) Europe - Tasks

(e) UK-2002 - Guided

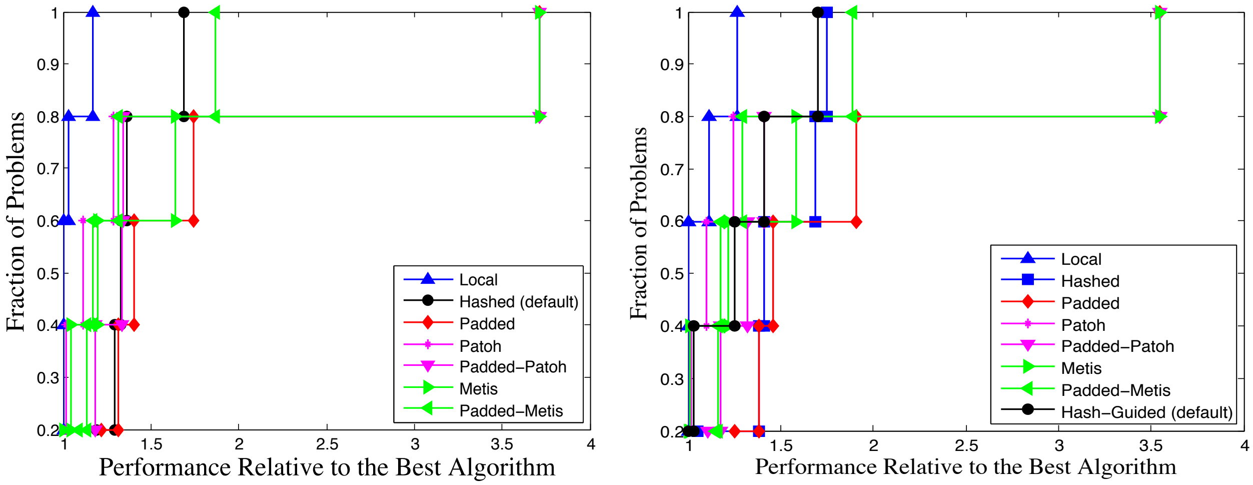
(f) UK-2002 - Tasks

Figure 4. **Strong scaling** results on Tilera for inputs listed in Table II. Figures on the left use guided scheduling, and those on the right use task stealing. On each input, results for different number of cores are shown. Speedups are relative to the serial algorithm of Blondel *et al.*. For the Europe data set, speedups are relative to the *hash* variant under guided scheduling since serial results were not available.

the U.S. Department of Energy Pacific Northwest National Laboratory (PNNL). PNNL is operated by Battelle Memorial Institute under Contract DE-AC06-76RL01830.

References

[1] U. Brandes, D. Delling, M. Gaertler, R. Grke, M. Hoefer, Z. Nikoloski, and D. Wagner, "Maximizing modularity is



(a) Performance profile - Guided

(b) Performance profile - Tasks

Figure 5. **Performance profiles** of different memory layout schemes for guided scheduling (left) and work stealing (right). When numbers do not exist for certain runs, they are replaced with large numbers.

hard," *arXiv preprint physics/0608255*, 2006.

[2] S. Gálvez, D. Díaz, P. Hernández, F. J. Esteban, J. A. Caballero, and G. Dorado, "Next-generation bioinformatics: using many-core processor architecture to develop a web service for sequence alignment," *Bioinformatics*, vol. 26, no. 5, pp. 683–686, 2010.

[3] R. Airoldi, F. Garzia, and J. Nurmi, "FFT algorithms evaluation on a homogeneous multi-processor system-on-chip." IEEE, 2010, pp. 58–64.

[4] V. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, p. P10008, 2008.

[5] S. Fortunato, "Community detection in graphs," *Physics Reports*, vol. 486, no. 3-5, pp. 75–174, Feb. 2010.

[6] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical Review E*, vol. 69, no. 2, p. 026113, 2004.

[7] H. Lu, M. Halappanavar, A. Kalyanaraman, and S. Choudhury, "Parallel heuristics for scalable community detection," in *Proc. International Workshop on Multithreaded Architectures and Applications*, vol. In Press., Phoenix, AZ, 2014, p. 110.

[8] U. Çatalyürek, J. Feo, A. H. Gebremedhin, M. Halappanavar, and A. Pothen, "Graph coloring algorithms for multi-core and massively multithreaded architectures," *Parallel Computing*, 2012.

[9] N. M. Josuttis, *The C++ Standard Library: A tutorial and reference*. Addison-Wesley Professional, 2012.

[10] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, Dec. 1998.

[11] U. Çatalyürek and C. Aykanat, "A hypergraph-partitioning approach for coarse-grain decomposition," in *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, 2001.

[12] D. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, "Graph partitioning and graph clustering: 10th DIMACS implementation challenge workshop," *Contemporary Mathematics*, vol. 588, Feb. 2012.

[13] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1–25, Dec. 2011.

[14] M. E. J. Newman, "Analysis of weighted networks," *Phys. Rev. E*, vol. 70, no. 5, p. 056131, Nov. 2004.

[15] A. Clauset, M. E. J. Newman, and C. Moore, "Finding community structure in very large networks," *Phys. Rev. E*, vol. 70, no. 6, pp. 66–111, Dec. 2004.

[16] K. Wakita and T. Tsurumi, "Finding community structure in mega-scale social networks:[extended abstract]." ACM, 2007, pp. 1275–1276.

[17] E. J. Riedy, H. Meyerhenke, D. Ediger, and D. A. Bader, "Parallel community detection for massive graphs," in *Parallel Processing and Applied Mathematics*. Springer, 2012, p. 286296.

[18] B. Auer and R. Bisseling, "Graph coarsening and clustering on the GPU," in *10th DIMACS Impl. Challenge*, Atlanta, GA, 2012.

[19] S. Bhowmick and S. Srinivasan, "A template for parallelizing the louvain method for modularity maximization," in *Dynamics On and Of Complex Networks, Volume 2*. Springer, 2013, pp. 111–124.

[20] C. L. Staudt and H. Meyerhenke, "Engineering high-performance community detection heuristics for massive graphs." IEEE, 2013, p. 180189.

[21] D. Jagtap, K. Bahulkar, D. Ponomarev, and N. Abu-Ghazaleh, “Characterizing and Understanding PDES Behavior on Tilera Architecture,” in *Proceedings of the 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*, 2012, pp. 53–62.

[22] S. Safari, A. Fijany, F. Diotalevi, and F. Hosseini, “Highly parallel and fast implementation of stereo vision algorithms on MIMD many-core tilera architecture.” IEEE, 2012, pp. 1–11.

[23] Y.-F. Hung, S.-Y. Tseng, C.-T. King, H.-Y. Liu, and S.-C. Huang, “Parallel implementation and performance prediction of object detection in videos on the tilera many-core systems.” IEEE, 2009, pp. 563–567.

[24] A. Morari, A. Tumeo, O. Villa, S. Secchi, and M. Valero, “Efficient Sorting on the Tilera Manycore Architecture,” in *24th IEEE International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Oct 2012, pp. 171–178.

[25] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele, “Many-core Key-value Store,” in *Proceedings of the 2011 International Green Computing Conference and Workshops*, ser. IGCC ’11, 2011, pp. 1–8.