DFGR: an Intermediate Graph Representation for Macro-Dataflow Programs

Alina Sbîrlea
Rice University
alina@rice.edu

Louis-Noël Pouchet
University of California Los Angeles
pouchet@cs.ucla.edu

Vivek Sarkar
Rice University
vsarkar@rice.edu

***Abstract*—In this paper we propose a new intermediate graph representation for macro-dataflow programs, DFGR, which is capable of offering a high-level view of applications for easy programmability, while allowing the expression of complex applications using dataflow principles. DFGR makes it possible to write applications in a manner that is oblivious of the underlying parallel runtime, and can easily be targeted by both programming systems and domain experts. In addition, DFGR can use further optimizations in the form of graph transformations, enabling the coupling of static and dynamic scheduling and efficient task composition and assignment, for improved scalability and locality. We show preliminary performance results for an implementation of DFGR on a shared memory runtim system, offering speedups of up to 11× on 12 cores, for complex graphs.**

I. Introduction

It is now well understood that extreme-scale computing will be faced by key software challenges, including those related to concurrency, energy, and resilience [1]. These challenges are prompting the exploration of new approaches to programming and execution systems, and, specifically, re-visiting of the dataflow model to find new ways to address the challenges of extreme-scale software. In the early days of dataflow computing, there was a belief that new programming languages such as VAL [2], Sisal [3], and Id [4] were necessary to obtain the benefits of dataflow execution. However, there is now an increased realization that "macro-dataflow" execution models [5] can be supported on standard multi-core processors by using data-driven runtime systems [6]–[8]. There are many benefits that follow from macro-dataflow approaches, including simplified programmability, increased asynchrony, support for heterogeneous parallelism, and scalable approaches to resilience. As a result, a wide variety of programming programming systems have begun exploring the adoption of dataflow principles, ranging from new programming models such as Concurrent Collections (CnC) to the new task dependency feature in OpenMP 4.0 [9]. Hence, there is now a growing need for compiler and runtime components to support macro-dataflow execution in these new programming systems.

In this paper, we introduce an intermediate graph representation for macro-dataflow programs that we call DFGR, and describe both its specification and implementation details. Our goal with DFGR is to enable its use by a wide range of programming systems. We describe how the DFGR model can be used as an abstraction to map applications for extreme scale systems, while remaining runtime independent. This allows such applications to run anywhere the underlying data-driven runtime can run, such as heterogeneous architectures including GPUs and FPGAs or distributed-memory clusters and data centers. DFGR can also help improving productivity by focusing the user on easily expressing task and data parallelism in a dynamic single assignment memory model. This makes it possible to perform numerous static and dynamic analyses and transformations of the macro-dataflow graph. The contributions of this paper are as follows.

- Learning from our experience with CnC [10] and the modeling and mapping of medical imaging applications in the CDSC project [11], we introduce DFGR, a dataflow graph representation language to model full applications and ease their deployment on homogeneous/heterogeneous parallel architectures.
- The DFGR representation can be easily implemented using task-level parallelism. We demonstrate a fully automatic translation system to Habanero-C, enabling DFGR graphs to be automatically executed anywhere the HC runtime operates, which includes shared-memory multicore, heterogeneous nodes containing GPUs and FPGAs, as well as distributed-memory clusters.
- DFGR subsumes several dataflow models, and can serve as a common representation for a wider range of dataflow programming systems.

II. The DFGR model

A. Macro-Dataflow for High-Performance Computing

A major objective of the Center for Domain-Specific Computing [11], [12] is to ease the development and deployment of applications on heterogeneous devices. To achieve this goal, we implement a two-level programming principle. Given a complete application, for instance an end-to-end medical imaging pipeline for CT scan reconstruction and analysis [11], [13], the application is first decomposed into steps, which can be executed atomically on a given device. A key feature to enable high-performance on a variety of devices is to allow the computation steps to be implemented in any language of choice, such as C/C++, CUDA, HC, etc. These step implementations form the low-level of the programming model. At the high-level, these steps may be called numerous times on different input data, and are seen as black boxes reading some data elements and producing some other data elements.

The coordination (control and data flow) between these steps is described through a DataFlow Graph Representation (DFGR). This graph is then automatically compiled into a parallel language, including the generation of all communications needed between step instances and all the data- and control-flow dependences between them, as described in the DFGR

978-1-4799-8095-6/14 $31.00 © 2014 IEEE
DOI 10.1109/DFM.2014.9

file. We can exploit a dynamic and heterogeneous run-time system for parallel step execution [13] that can be also ported to distributed computing [6], [14], by compiling a DFGR graph to a parallel language with good interoperability with other low level languages, such as Habanero-C. In our framework, *a single DFGR file is used* to model the full application data and control flow between these atomic steps, irrespective of the target platform used to execute the computation. Indeed, a key motivation for DFGR and its associated execution model is to decouple the task of *expressing* the parallelism, which should belong to the domain experts and algorithm designers, from the task of *implementing* the available parallelism on a given hardware. DFGR is meant to ease the expression of data flowing in and out of step instances by letting the user focus exclusively on modeling the application data and control characteristics without any concern about their actual implementation. But for this approach to succeed, it is required to ensure a set of properties of the language to allow for (1) easy debugging and analysis of the graph structure, for instance to detect races; (2) easy deployment on a variety of hardware, ranging from heterogeneous multi-core/GPUs systems to distributed computing on cluster; and (3) easy modeling of complex applications, including at different parallelism grains.

To address these three problems, we present DFGR, a macro-dataflow graph representation that builds on past work on Intel Concurrent Collections [10] and CnC-HC projects [13]. In a nutshell, DFGR is a macro-dataflow language that enforces dynamic single assignment for data, thereby greatly simplifying debugging and analysis. DFGR is fully and automatically translatable to the parallel language chosen for the concrete implementation of the model. To demonstrate this, we implemented a compiler from DFGR to the Habanero-C (HC) task-parallel language, which allows the graph to execute on any hardware implementing the HC runtime or equivalent [6], [10], [13]. Finally DFGR uses simple concepts such as unique tags to model the relationship between dynamic instances of steps and the particular data elements they access in a simple text representation, thereby greatly facilitating the description of the application's parallelism and data flow.

B. Key Features of DFGR

DFGR is meant to serve as a general representation which can rely on a parallel and distributed runtime for performance. Previous models such as Kahn Process Networks (KPN) [15] or the Synchronous Dataflow Model (SDF) [16] target streaming applications, while Concurrent Collections (CnC) primarily targets task parallelism. DFGR can express both the streaming model and task parallelism and it can be further optimized for both through analysis of the graph structure.

DFGR simplifies the CnC model by allowing dependencies to be expressed between steps and items, and also directly between steps and steps. It enhances CnC with a means of expressing precisely what items are read and written by each step. This is achieved through tag functions [13] (the tag identifying an item read by a step is a function of the step's tag) and regions modeling sets of tags. It also preserves the concept of affinities [13] between steps and underlying available hardware, allowing automatic tuning of the application, when multiple code variants for accelerators are made available.

A hierarchy of concepts for modeling sets of tags is provided, from simple ranges (e.g., rectangles) to affine integer tuple sets (e.g., polyhedra) to union of integer sets and finally arbitrary sets. This clear hierarchy allows for static analysis and optimization of the graph using compiler frameworks such as the polyhedral model, when the tag sets and relations between them are affine forms. These concepts are novel and key to the DFGR model; they are further detailed in Section III-C.

While in the streaming model instances of (e.g, calls to) steps run in sequence, in DFGR any instances of steps with no dependence between them are viewed as parallel tasks. Steps in the same collection may run on different cores or even different devices. With tools capable of discovering part of the graph topology statically, DFGR can make use of static scheduling, while the remaining graph topology will be discovered at runtime and it will rely on dynamic scheduling. CnC relies entirely on dynamic scheduling while models such as KPN and SDF expect the full graph topology to be known apriori. DFGR natively encompasses both.

Steps in DFGR are functional (e.g., stateless), so data accessed by steps is stored separately. All resources internal to the step are cleared when the step instance finishes. Data read by a step may come from any source, but any ⟨tag, token⟩ pair is only written once. Graph items are Dynamic Single Assignment (DSA) and are never collected, i.e. they are persistent. Optimizations such as folding the data space when an ordering is imposed or setting a fixed number of reads per item ("get_counts") can allow items to be collected for better storage management [17]. DFGR also permits arbitrary access to data outside of the graph: such global data can be used to create a state without using an item collection. However using global data limits the outcome of graph analysis and transformations, since this non-graph data is not represented nor analyzed in the macro-dataflow graph.

The creation of steps is achieved directly by one step spawning another step. This is different from CnC, where control (tag) collections are used for spawning steps, but also from streaming models where steps start as soon as data is available without the ability to choose which steps should run. Steps are allowed to make conditional Puts/Prescribes for expressing general applications, while no switch and select nodes are present in SDF. In DFGR, steps may also pass parameters to the steps they spawn.

A step can explicitly request to wait on another step in DFGR, while synchronization in previous models (including CnC) required indirect coordination via items. Inter-step synchronization in DFGR can be used to coordinate/schedule accesses to data that is not modeled in the DFGR graph through item collections. DFGR also allows non-determinism through constructs such as "PutIfAbsent", a method which only writes an item if it was not previously written. Graph traversals can use such constructs in order to establish the first visitor for a node.

Finally, a DFGR graph is *fully executable* without having to execute any of its actual step implementation. That is, the entire dynamic parallelism and data flow can be uncovered using a simple interpretation of the DFGR program, allowing for subsequent analysis and optimizations such as race detection and storage optimization.

III. DFGR LANGUAGE SPECIFICATION

A. Core Features for Macro-Dataflow Modeling

DFGR is a graph representation that contains two main components: *steps*, that represent pieces of computation; and *items*, that represent pieces of data read and written by steps. The user describes an application by writing a graph (in textual form or using an API to create the graph) that captures the relation between data items and steps. In order to model *explicitly* all the dynamic instances of each step as well as all items during the execution of the application modeled, both steps and items are grouped into collections within which they have unique identifiers called *tags*. In order to guarantee the graph is deterministic and free of data races, all data in item collections must follow the dynamic single assignment rule, that is an item in a collection is never written more than once.

An item collection is a group of data items having the same type. Each item in the collection can be uniquely identified by its tag, thus an item collection is a set of (tag, value) pairs. Items can be written to a collection by the environment and also by other steps. Similarly, items can be read by steps and by the environment once the graph execution has finished. Item collections are declared in the textual representation using brackets: `[int* A]` declares a collection of items which are pointers to integers. Using a pass-by-value mechanism, any type, including structures and arrays, can be used for items.

The human-friendly modeling of all data elements being read and/or written by a step instance is achieved by relating the tags of item collections with tags associated to step instances. For instance `[A : i]` models tag `i` of collection `A`. Then `[A : i-1] -> (S : i) -> [A : i]` models that instance `i` of `S` will read element `i-1` of collection `A`, and produce element `i`. In DFGR there are multiple ways to describe tags, as discussed in Sec. III-C. In its most general form the user can write `[A : foo(...)]` to describe a tag value, where `foo` is a call to some pure function possibly requiring run-time evaluation to compute its value.

A step collection is a group of instances of the same step. The unique identifier (tag) of a step instance can carry semantics used by the step implementation itself, for instance the tag can behave like a surrounding loop iterator. Steps can be started by the environment which is in charge of initializing and starting the graph, and also by other steps. Depending on the model's implementation, it can adhere to the strict preconditions model, where steps will not execute until all its input data is made available; steps can execute eagerly and block or rollback when data is not available; or have a flexible approach through the flexible preconditions model [18]. Steps are written using parentheses: `(S)`. DFGR uses arrows to express reads and writes: `[A]->(S)` and double colon to express the creation of new steps: `(S1)::(S2)`. When using tags, the notation `(S : i)` models instance `i` of step `S`, and `env::(S : {1..42})` models that the environment `env` will prescribe at start 42 instances of `S`, that is `i` will range from 1 to 42. The modeling of data and control dependences between step instances is achieved through the modeling of the data read/written by a step instance, and also using an explicit step-to-step (e.g., point-to-point) synchronization construct. For instance `(S1 : i) -> (S2 : i)` models that instance `i` of step `S2` will not start until instance `i` of step `S1` completed.

B. Example: Smith-Waterman in DFGR

In this section we take the Smith-Waterman sequence alignment algorithm and show the steps needed to write an application in DFGR. The DFGR representation can originate from hand-written user code, from tools analyzing dependences in sequential programs or from other graph representations.

Writing a DFGR representation implies that the user must reason about the computation that exists within the graph, the data read and written and how this information flows from one step to another. In Figure 1, we give a visual representation of the computation performed on a matrix in the Smith-Waterman algorithm. We identify 4 kind of steps: a *single* step (S) computing the top-left matrix corner, and a set of steps computing the *top* row (T), *left* column (L) and the *center* (C) of the matrix. The arrows mark the flow of data, e.g. the information from step (S) is read by three other steps (T),(L) and (C), while each step (T) provides input to another instance of step (T) and 2 instances of step (C). In this example it becomes clear the need to group steps into collections and use unique identifiers to differentiate between instances of the same step. Let us assume that we are using a NH $\times$ NW matrix. Then, there are (NH-1) $\times$ (NW-1) center steps, where each can be identified by a unique tag (i,j), with $1 \leq i \leq NH$ and $1 \leq j \leq NW$. From Figure 1 we can also infer data dependences, e.g., all center steps read 3 items and write a single item. Using the tuple (i,j) as the unique tag identifer, we can say that each step (C:i,j) reads items [A:i-1,j-1], [A:i-1,j], [A:i,j-1] and writes [A:i,j].

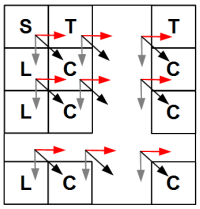

Fig. 1: Smith-Waterman: The computation steps are grouped in a matrix structure based on their unique identifiers (i,j) and the items they write [A:i,j]. Arrows show data dependences for each step.

Alternatively, a graph representation can originate from automatic analysis of a sequential code such as in Listing 1. In this code snippet we abstracted the actual computation performed by each step with a function call. Note that from this code we can also infer the dependences specified before, in particular what items each step reads and writes and a unique identifier for each step. As it is required to use the dynamic single assignment form for DFGR, if the input code is not in DSA form already a promotion to DSA must be performed during the translation to DFGR.

The DFGR file for Smith-Waterman in shown Listing 2. The first line of code declares an item collection, where each item is of type int. The next four lines of code specify for each of the 4 steps what items are read and written, using the unique tags for both steps and items. The final four lines specify what the environment needs to produce for the graph to start, and what it needs to emit after completion of the graph (output data). The environment will start all computation steps and it

Listing 1: Sequential Smith-Waterman code.

```
A[0][0] = single_corner();
for(j=1; j<NW; j++)
   A[0][j] = top(j);
for(i=1; i<NH; i++) {
   A[i][0] = left(i);
   for(j=1; j<NW; j++)
      A[i][j] = center(i, j, A[i-1][j-1],
                       A[i-1][j],A[i][j-1];
}
```

will read one item resulting from the computation (the bottom right corner, the sequence alignment cost in Smith-Waterman).

Listing 2: DFGR for Smith-Waterman.

```
[int A];
      //Steps' I/O relations
(single_corner:i,j) -> [A:i,j];
[A:i,j-1] -> (top:i,j) -> [A:i,j];
[A:i-1,j] -> (left:i,j) -> [A:i,j];
[A:i-1,j-1], [A:i-1,j], [A:i,j-1] ->
        -> (center:i,j) -> [A:i,j];
     //Steps started by the environment.
env::(single_corner:0,0);
env::(top:0,{1 .. NW+1});
env::(left:{1..NH+1},0);
env::(center:{1..NH+1},{1..NW+1});
[A:NH,NW] -> env;
```

C. In-Depth: DFGR Tags

1) Tag Functions Explained: We say that a I/O and prescription rules are composed of two parts connected by the $->$ and :: operators respectively. One side is the step driver, declared between parentheses and identified by a multi-dimensional tag, abstracted by user-chosen variable names. The other side (left or right for I/O relations and right for prescriptions) is a list of items or steps each identified by a list of functions, where each such function is a function of the driver step's tag components. The diagram in Figure 2 gives an example of a step prescription and aims to clarify what are tags, tag components and tag functions.

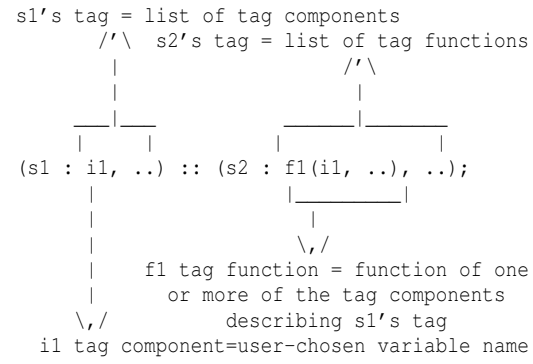

Fig. 2: Step tags, tag components and tag functions explained

2) A Hierarchy of Concepts for Tag Sets: A key feature of DFGR is to attempt to reconcile static and dynamic macro-dataflow modeling, especially by allowing the explicit and unique naming of each step instance and each item in a data collection. To achieve this goal we propose a hierarchy of concepts that can be used to represent sets of tags, ranging from the simplest form of integer ranges to the most complex form of arbitrary sets. For instance the example in Listing 2 uses *ranges* to describe the step instances prescribed by the environment, i.e., {`1..NW+1`}. But in the general case, more complicated shapes can be required to properly model the set of steps. On the other hand, allowing for abitrary sets without any property allow to model the general case, but may dramatically limit the static analysis that can be performed.

The motivation for introducing several different concepts in a structured way relates to the different kinds of static analysis frameworks that could be used at the graph level. Limiting to particular constructs to model tag sets will automatically enable or disable certain static analysis frameworks. Focusing only on using integer tuples for the tags, for good expressiveness we need to capture the sets of tags to be used in the representation, be it for describing a set of step instances, a set of prescribed steps, or a set of data being read/written by a step. We propose the following hierarchy for these sets: *(1) ranges; (2) simple polyhedron; (3) union of $\mathcal{Z}$-polyhedra; (4) union of arbitrary sets*.

In a nutshell, ranges will model rectangles and are well suited for simple, regular computations. Polyhedra, which are described using affine inequalities, enable powerful static analysis and transformation of the graph based on the affine framework [19], [20]. Unions of $\mathcal{Z}$-polyhedra are a generalization of polyhedra, allowing for more complex geometric shapes to be modeled by using union of convex sets, and regular strides in the set are supported using affine lattices [21]. Modern polyhedral compilation frameworks fully support the complete analysis of programs described using such polyhedra [22]–[24]. Finally, arbitrary sets are sets not belonging to any of the previous category, typically used when the set cannot be described using strides and affine inequalities; or when the set is described using functions whose result is not statically known, for example `foo(i,j)` where `foo` is seen as an uninterpreted function at the graph level. In our hierarchy, the less expressive the set type is, the simpler it is to implement static analyses and code generation to a parallel language on such sets.

a) Ranges and Simple Polyhedra: The role of tag sets is analoguous to the role of iteration spaces for step prescription, and data sets for items. Tag sets can be used to express complex I/O and prescription dependences in tag functions.

In the code snippet below we use a 1-dimensional range to define that step $s1$ may write $i1$ items into $item1$ with tags ranging from 0 to $i1$.

```
(s1 : i1) -> [item1 : {0 .. i1}];
```

A key benefit of ranges is that they are very simple to analyze and translate. For instance one can scan a set described by a range using a simple `for` loop, using the bounds of the range as the loop bounds. Multidimensional ranges are simply scanned using nested for loops.

The same set can also be defined using a *region*, the generic name in DFGR for tag sets which are not ranges. Polyhedra, union of $\mathcal{Z}$-polyhedra and arbitrary sets are all considered regions in DFGR. A basic syntactic analysis of the region expression is enough to determine which actual type of set is being implemented. Here is an example of defining and using a region $reg1$:

```
<reg1(ub) : r1> { 0 <= r1, r1 <= ub };
(s1 : i1) -> [item1 : r1; reg1(i1)];
```

The first line defines a region named `reg1`, which is a 1-dimensional set. A parameter (that is, an unknown constant) is used in the region definition: `ub`. `r1` is the variable associated to this set, that is, `r1` will take all values defined by the set `reg1`. The region is defined here using a conjunction of affine inequalities each separated by `,` so it is a basic polyhedron. The second line is an I/O relation which uses the region $reg1$. It says that a step instance – identified by $i1$ – from step collection $s1$, will write all items with tag $r1 \in reg1$. We can combine tag functions with regions, such as in:

```
(s1 : i1) -> [item1 : f1(r1); reg1(i1)];
```

which models that instance $i1$ of $s1$ may write into item collection $item1$, for all $r1 \in reg1$, any or all values with tags given by the tag function $f1(r1)$.

Ranges may be used to describe *unnamed, rectangular, contiguous* and *multidimensional* iteration spaces.

```
(s1:i,j)->[item1:{i-1..i+1},{i+j..i*j+1}];
```

In the example above, there are two ranges, one for each of the tag components of the item collection, we also say it defines a 2-dimensional range. First, note that ranges are *unnnamed*. Secondly, the sets defined by ranges are *rectangular*, because without names each dimension is independent of the others (i.e. the second range cannot refer to an particular item in the first range). As an analogy with for loops, ranges cannot express a triangular loop nest. Thirdly, the ranges can only be used at the outer level when specifying a tag function. To be more precise, it is correct to say "$\{i+1..i+5\}$" but incorrect to say "$i+\{1..5\}$". In this example the two notations are equivalent, because addition (and subtraction) is distributive with a range. However multiplication is not distributive, nor is division. As such, there is no means of specifying that a step writes for example items $i*2, i*3, i*4...i*N$ using a range, but we can easily write this with a region. We say thus that a range is a *contiguous* set of integer points, whereas a region need not be.

b) More Region Constructs: In practice, ranges and simple polyhedra are often enough to express the tag sets needed to model an application. They also come with the benefit of easy compilation to a loop-based language, as scanning such sets is straightforward for ranges (see above) but also for simple polyhedra after normalizing the inequalities in row-echelon form. On the other hand, more complicated patterns may need to be expressed. In our taxonomy for tag sets, we purpotedly isolated union of $\mathcal{Z}$-polyhedra from arbitrary sets because, similarly to ranges and simple polyhedra, *there exists readily available tools to generate code scanning those sets at compile-time*. For instance CLooG [23] and ISL [24] both generate automatically code scanning those sets using only `for` loops and `if` conditionals, making the translation of these sets to the target parallel language straightforward. We remark that numerous static analysis (dataflow analysis, scheduling, etc.) are also available for programs described using only this type of set.

In its general form, a region is a union (e.g., disjunction) of conjunction of inequalities. Its inequalities may be a function of parameters and uninterpreted functions. For instance the region below:

```
<reg2(i,j):r1,r2>{i-1<=r1,r1<=i+1,i+j<=r2,r2<=i*j+1};
```

is a conjunction of four inequalities, i and j are parameters, and $r1$ and $r2$ are the two dimensions of this set, e.g., points in this set are tuples $(r1, r2)$. As i and j are parameters, this set is actually a simple polyhedron. The region below:

```
<reg3(i,j):x,y>{i<=x,x<=j,y>=0,y<=fc(x,j)},{x=1,y=3};
```

represents a union of sets (separated by `,`), where in the first part an uninterpreted function `fc(x,j)` is used. Without further information on this function, this set cannot be analyzed statically using polyhedral frameworks, thereby limiting the analysis that can be performed on the whole graph. On the other hand, it allows the modeling of arbitrary sets, but at the expense of a possibly costly run-time scanning code as the function needs to be evaluated for each x value.

IV. Implementation Details

A. Language tool chain

The high-level DFGR model is designed to be language independent, and the concrete implementations of the model will make the language choice. We created an implementation of DFGR that relies on a parallel C programming model, in order to achieve both performance and interoperability with languages dedicated to specialized hardware.

Figure 3 presents the implementation flow for an application written in DFGR.

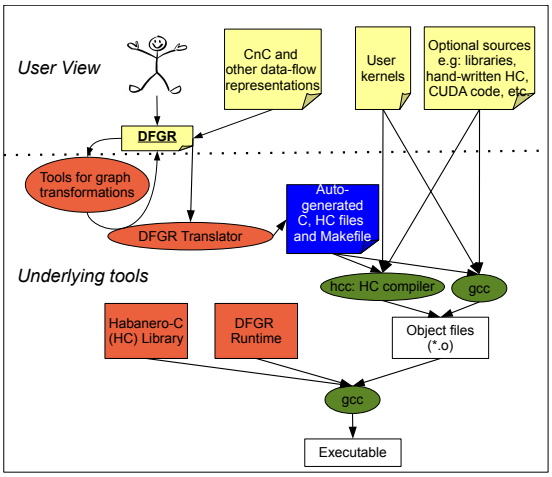

Fig. 3: DFGR Implementation Flow

The steps for creating an application using DFGR are the following. First, an application needs to be written in the DFGR format, i.e. the algorithms needs to be decomposed into steps, along with the producer-consumer relationships between them. This textual graph can be either written by the

user, or it can be generated by tools analyzing and translating from another programming language, or from other dataflow modeling using a similar graph representation. An example of the latter is generating DFGR from CnC, but note that this translation cannot use all the features in DFGR, as CnC's specification is more restrictive.

Further, we provide a translator that generates a series of "glue-code" files to enable transparent parallelism for the user, using the inter-step dependences provided in the graph file. The translator also creates code stubs for each of the computation steps, and offers guidelines for creating the user code, i.e., for writing items and spawning new steps. The translator also provides a "full-auto" mode, which assumes all items declared by a step in a graph will be written and all steps prescribed and creates fully compilable steps instead of the step stubs. The user can then simply add his own file containing the implementation of the computation steps, without ever seeing any of the API needed by DFGR or its translation to HC.

Once the user has provided the code for the computation steps, they can use the makefile generated by the translator to build the application. The build system uses the Habanero C compiler (see Section IV-C) and the gcc compiler to generate an executable file. If additional libraries are required, they can be easily added in the provided makefile.

B. Implementing item collections

For efficiency, we implement item collections in DFGR using a hash-table for each collection, an approach similar to previous implementations of the CnC model. Indexing in the hash-table is done via the key which uniquely identifies each item in the collection.

As items must adhere to the DSA rule, we have created safety nets to warn users an item with the same tag is written multiple times. We do however allow items to be cleared from a collection once they are no longer used. This information can provided by the user when writing to a collection, by specifying a "get-count", which is a number equal to the number of times the item will be read. If this information is not specified, the item will remain in the collection for the duration of the graph execution.

C. Implementing step collections

Steps are asynchronous pieces of computation, that we implement using a task-parallel programming model: Habanero-C [25]. A collection of steps is a theoretical construct, which refers to the fact that these computations perform the same function, and their only side effect are the items they write and the new steps they transitively create.

The Habanero-C (HC) language is a C-based task-parallel programming language developed at Rice University. In this section, we summarize key properties of HC and the Habanero model as described in [25]–[27]. The two main features of HC that are relevant to this paper are the *async* and *finish* constructs, which define lightweight dynamic task creation and termination and were originally defined in X10 [28].

A new child task is created using statement "**async** ⟨*stmt*⟩". This will run ⟨*stmt*⟩ *asynchronously* (i.e. before, after, or in parallel) with the remainder of the parent task. Figure 4 provides a diagram in which the parent task, T_0, uses an `async` construct to create a child task T_1. Thus, `STMT1` in task T_1 can potentially execute in parallel with `STMT2` in task T_0.

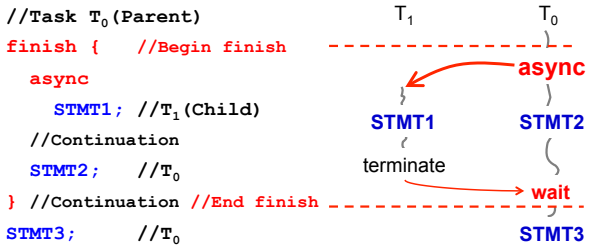

Fig. 4: An example code schema with `async` and `finish` [27]

`async` is a powerful primitive because it can be used to enable any statement to execute as a parallel task, including statement blocks, for-loop iterations, and function calls. In this work we use the `async` statement to create dynamic instances of DFGR steps.

`finish` is a generalized join operation. The statement "**finish** ⟨*stmt*⟩" causes the parent task to execute ⟨*stmt*⟩ and then wait until all transitively spawned tasks using the `async` primitive have completed.

Each dynamic instance T_A of an `async` task has a unique *Immedia Enclosing Finish* (IEF) instance F of a `finish` statement during program execution, where F is the innermost `finish` containing T_A [29]. There is an implicit `finish` scope surrounding the body of `main()`, so program execution will only end after all `async` tasks have completed.

For example, the `finish` statement in Figure 4 is used by task T_0 to ensure that child task T_1 has completed executing `STMT1` before T_0 executes `STMT3`. If T_1 created a child `async` task, T_2 (a "grandchild" of T_0), T_0 will wait for both T_1 and T_2 to complete in the `finish` scope before executing `STMT3` [13]. We only use one level of `finish` when executing a DFGR program because we include additional *data-driven* execution constraints on `async` tasks that control when `async` tasks become ready for execution.

D. DFGR runtime

The implementation of the DFGR model we present in this paper relies on a runtime built on top of the Habanero-C programming language. It uses the async construct to create asynchronous running steps and the finish construct to wait for the termination of the graph. The DFGR runtime implementation involves ensuring that steps are only started when all the data they need for execution is available. This is achieved by testing the satisfiability of all dependences before each steps's execution and queuing up the steps on the missing data. In the DFGR runtime, the creator C of a data item takes the responsibility of checking for waiting steps. If the current data is the last one in the waiter's dependency list, C sends the new steps to the scheduler for execution. Alternatively, it will reenqueue the step on the next missing piece of data it needs. Note that since DFGR is designed as a high-level representation of data-flow application, its concrete implementation can vary. The implementation we presenting this paper can be used as a reference for future works.

V. Preliminary results

Our results were obtained on a 12 core Intel Xeon CPU X5660 @ 2.80GHz. The results presented here are an average of 5 runs for tiled implementations of a series of benchmarks (the implementation is C based with a standard deviation below 5%). We remark we did not perform any tile size tuning or any optimization of the scheduling/placement of step instances in these preliminary experiments. These result are but a guideline, showing reasonable performance on the CPU. The underlying runtime relies on the work presented in [13] and, as such, exhibits similar performance with it. The scope of this paper is an overview of DFGR, so we do not show results on heterogeneous architectures; the current implementation has this capability, using similar constructs presented in [13].

- *Smith-Waterman* is an algorithm from the biology domain, which performs sequence alignment between two strings or nucleotide or protein sequences. We run the alignment algorithm for 2 strings of size 50000 each, with a tile size of 400 in each dimension.
- *Black-Scholes* is a financial application that computes stock values. We use an input size of 1,500,000 and granularity 128.
- *Cholesky Factorization* is a linear algebra benchmark that decomposes a symmetric positive definite matrix into a lower triangular matrix and its transpose. The input matrix size is 3000 $\times$ 3000 and the tile size is 150$\times$150.
- *Matrix Inverse* creates the inverse of a symmetric positive definite matrix using an algorithm which decomposes the input matrix into tiles and computes partial matrices before obtaining the inverse. The input matrix size is 4096 $\times$ 4096 and the tile size is 64 $\times$ 64.
- *Denoise* is an algorithm for processing 3D images in order to remove noise. We use an input of 256 $\times$ 256 $\times$ 256 and a tile size of 32 $\times$ 32 $\times$ 32.

For all benchmarks we use the strict preconditions model, for performance [18], which models applications where all step dependences are known in advance – outlined in a DFGR format in our benchmarks – and steps only start where all inputs are available. We see in Figure 5 that all benchmarks described above have already good strong scaling results, even without tuning the tile sizes or applying high-performance optimizations to the computation kernels. In addition, the DFGR model offers great user productivity due to the automatic generation of the code glueing together a parallel application, relying on the user only to describe the algorithm in DFGR and to provide the computation kernels.

VI. Related work

DFGR is a data-flow model and by definition exposes an application's available parallelism. Traditional programming models express sequential computations, or provide complex primitive for creating parallel programs. This either limits the user's capability of expressing arbitrary parallel computations or requires complex knowledge and understanding or parallel models. In contrast, the data-flow model enables the expression of intrinsically parallel applications through the decomposition of applications into tasks and their relations.

DFGR has its roots in Intel's Concurrent Collections (CnC) [10], [30], a macro-dataflow models which takes a similar

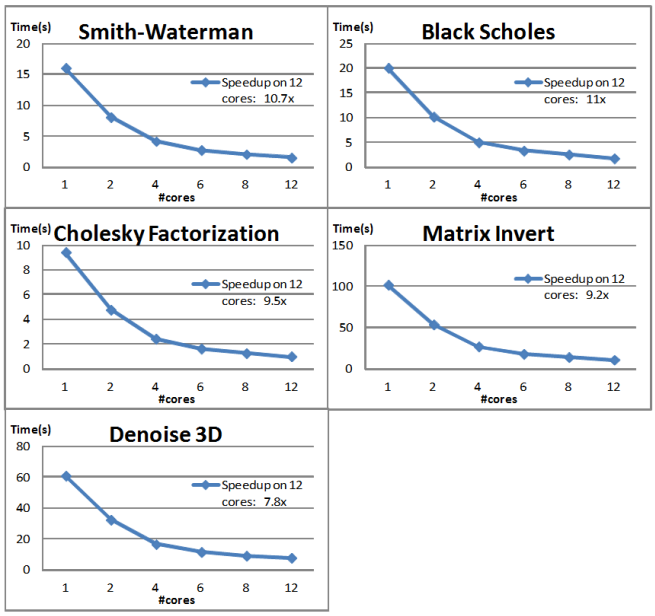

Fig. 5: Scalability results for 5 benchmarks implemented using DFGR

approach providing a separation of concerns between a domain expert, who can provide an accurate problem specification at a high level, and a tuning expert, who can tune the individual components of an application for better performance. In the original CnC implementation however there was no means of knowing which item from a particular item collection a step should read, so such information needed to be given by the user inside the step code. The CnC model was extended [13] to enable accurate definition of dependences at a high level, in a textual graph specification, removing the need to make a user familiar with a particular API. DFGR takes the idea of a high-level data-flow model one step further, by simplifying the way of specifying step-item and step-step relations one the one hand, and offering more complex means of describing complex applications on the other. DFGR is also designed to be a possible standard for an intermediate representation of parallel programs in a macro-dataflow setup. Another motivation for this approach is to reuse the optimizations performed at the graph level, such as granularity adjustments, graph unrolling and static mapping of partial graphs onto low level architectures.

In this work we use the Habanero-C language to express parallelism. Other data-flow models take a similar approach of using either a threading library, such as pthreads used in TFlux [31], or a task library, such as TBB used in Intel's CnC, or a parallel language such as Cilk used in Nabbit [32]. As with Cilk, Habanero-C relies on a work-stealing scheduler [33], which, when used with arbitrary task graphs can overload the machine. DFGR aims to ease addressing this problem by separating the application description from its concrete implementation, for instance by enabling graph transformations to coarsen the granularity of parallelism.

One of the most commonly used parallel language is

OpenMP. In this model it was shown that having synchronizations using barrier is possibly very limiting for expressing applications. Recent work [34] has extended the OpenMP model to enable do-across parallelism for point-to-point synchronization, in order to make synchronization more efficient and enable data-locality.

Legion [35] is another language which aims to ease the programmer's use by taking as input a sequential program and automatically determining the dependences by creating a complete set of read-write statements for all data and enabling paralellism when no dependences are found. This model however requires an initial sequential specification of a program, while DFGR does not. In addition DFGR could benefit from the locality optimizations performed in Legion.

VII. Conclusions

In this paper we proposed DFGR, a graph representation for macro-dataflow programs offering high programmability for exascale computing. We outlined the language features that enable DFGR to model complex applications and the potential for graph analysis for DFGR applications. We have shown preliminary results of an implementation of the DFGR model, which offers good scalability for complex graphs. Our ongoing work involves taking the concepts presented in this paper and performing graph transformations to coarsen task granularity. Together with future transformations on the graph representation, these can lead to achieving performance, locality and ease of mapping onto heterogeneous hardware.

Acknowledgment

This work was supported in part by the Center for Domain-Specific Computing (CDSC) funded by NSF "Expeditions in Computing" award 0926127.

References

[1] V. Sarkar, W. Harrod, and A. E. Snavely, "Software Challenges in Extreme Scale Systems," January 2010, special Issue on Advanced Computing: The Roadmap to Exascale.

[2] W. Ackerman and J. Dennis, "VAL - A Value Oriented Algorithmic Language," MIT Laboratory for Computer Science, Tech. Rep. TR-218, June 1979.

[3] J. McGraw, *SISAL - Streams and Iteration in a Single-Assignment Language - Version 1.0*. Lawrence Livermore National Laboratory, July 1983.

[4] Arvind, M. Dertouzos, R. Nikhil, and G. Papadopoulos, "Project Dataflow: A parallel computing system based on the Monsoon architecture and the Id programming language," MIT Lab for Computer Science, Tech. Rep., March 1988, computation Structures Group Memo 285.

[5] V. Sarkar and J. Hennessy, "Partitioning Parallel Programs for Macro-Dataflow," *ACM Conference on Lisp and Functional Programming*, pp. 202–211, August 1986.

[6] "Building an open community runtime (ocr) framework for exascale systems," Nov. 2012, supercomputing 2012 Birds-of-a-feather session.

[7] "The Swarm Framework. http://swarmframework.org/."

[8] The STE——AR Group, "HPX, a C++ runtime system for parallel and distributed applications of any scale." [Online]. Available: http://stellar.cct.lsu.edu/tag/hpx

[9] OpenMP Architecture Review Board, "The OpenMP API specification for parallel programming, version 4.0," 2013. [Online]. Available: www.openmp.org/mp-documents/OpenMP4.0.0.pdf

[10] IntelCorporation, "Intel (R) Concurrent Collections for C/C++," http://softwarecommunity.intel.com/articles/eng/3862.htm.

[11] J. Cong, V. Sarkar, G. Reinman, and A. Bui, "Customizable Domain-Specific Computing," *IEEE Design and Test*, no. 2:28, pp. 6–15, Mar 2011.

[12] UCLA, Rice, OSU, and UCSB, "Center for Domain-Specific Computing (CDSC)," http://cdsc.ucla.edu.

[13] A. Sbîrlea, Y. Zou, Z. Budimlić, J. Cong, and V. Sarkar, "Mapping a data-flow programming model onto heterogeneous platforms," ser. LCTES '12.

[14] Habanero Research Programming Group, "CnC on the Open Community Runtime (OCR)," https://github.com/habanero-rice/cnc-ocr.

[15] G. Kahn, "The semantics of a simple language for parallel programming," *Jack L. Rosenfeld (Ed.): Information Processing 74, Proceedings of IFIP Congress 74*.

[16] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," in *Proceedings of the IEEE*, vol. 75, no. 9. IEEE, September 1987, pp. 1235–1245.

[17] D. Sbirlea, K. Knobe, and V. Sarkar, "Folding of tagged single assignment values for memory-efficient parallelism," in *Euro-Par'12*, 2012, pp. 601–613.

[18] D. Sbîrlea, A. Sbîrlea, K. Wheeler, and V. Sarkar, "The flexible preconditions model for macro-dataflow execution," ser. DFM, 2013.

[19] U. Banerjee, *Unimodular transformations of double loops*. University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, 1990.

[20] P. Feautrier, "Dataflow analysis of scalar and array references," *IJPP*, vol. 20, no. 1, pp. 23–53, 1991.

[21] G. Gupta and S. Rajopadhye, "The z-polyhedral model," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'07)*. ACM, 2007, pp. 237–248.

[22] D. K. Wilde, "A library for doing polyhedral operations," IRISA, Rennes, France, Tech. Rep. 785, 1993.

[23] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *PACT*, 2004, pp. 7–16.

[24] S. Verdoolaege, "ISL: An integer set library for the polyhedral model," in *Mathematical Software–ICMS 2010*. Springer, 2010, pp. 299–302.

[25] "https://wiki.rice.edu/confluence/display/HABANERO/Habanero-C."

[26] J. Cong, V. Sarkar, G. Reinman, and A. Bui, "Customizable Domain-Specific Computing," *IEEE Design and Test*, no. 2:28, pp. 6–15, Mar 2011.

[27] J. Payne, V. Cavé, R. Raman, M. Ricken, R. Cartwright, and V. Sarkar, "DrHJ — a lightweight pedagogic IDE for Habanero Java," in *PPPJ'11: Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java*, 2011.

[28] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," ser. OOPSLA '05.

[29] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer, "Phasers: a unified deadlock-free construct for collective and point-to-point synchronization," in *ICS '08*. ACM, 2008, pp. 277–288.

[30] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Taşirlar, "Concurrent Collections," *Scientific Programming*, vol. 18, pp. 203–217, August 2010.

[31] K. Stavrou, M. Nikolaides, D. Pavlou, S. Arandi, P. Evripidou, and P. Trancoso, "TFlux: A portable platform for data-driven multithreading on commodity multicore systems," ser. ICPP '08, 2008.

[32] K. Agrawal, C. Leiserson, and J. Sukha, "Executing task graphs using work-stealing," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, April 2010, pp. 1 –12.

[33] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, 1999.

[34] J. Shirako, P. Unnikrishnan, S. Chatterjee, K. Li, and V. Sarkar, "Expressing doacross loop dependencies in openmp," in *9th International Workshop on OpenMP*, ser. IWOMP, 2013.

[35] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12, 2012.