Hierarchically Tiled Array as a High-Level Abstraction for Codelets

Chih-Chieh Yang*, Juan C. Pichel†, Adam R. Smith*, and David A. Padua*
*Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL, USA
Email: {cyang49, smith195, padua}@illinois.edu
†CITIUS
Universidade de Santiago de Compostela
Santiago de Compostela, Spain
Email: juancarlos.pichel@usc.es

***Abstract*—The move from terascale to exascale systems is challenging in terms of energy and power consumption, resilience, storage, concurrency, and parallelism. These challenges require new fine-grain execution models to support the concurrent execution of millions or even billions of threads on the exascale machines. The most promising approaches are those based on the codelet execution model, which provide a flexible programming interface that allows the expression of all kinds of parallelism with fine-tuning opportunities. We propose using Hierarchically Tiled Array (HTA) as a high-level abstraction for codelets to improve the programmability and readability of programs while preserving the good performance and scalability provided by the codelet execution model.**

I. Introduction

The next challenge in the evolution of supercomputers will be the transition to exascale systems. At first glance, this goal seems reachable assuming Moore's law continues to hold. However, while the move from terascale to petascale processing was evolutionary, the leap to exascale supercomputers will require revolutionary advances. Simply scaling up the current technology will not work. The projections for the exascale systems indicate that applications may have to support up to a billion separate threads to efficiently use the hardware, while the amount of memory per functional unit will drop significantly [1], [2]. This implies the need for exploiting fine-grain parallelism which requires a programming model other than the dominant message passing paradigm which makes use of coarse-grain threads.

The codelet execution model [3]–[5] is a promising alternative to the message passing paradigm. The codelet model is a fine-grain program execution model inspired by the earlier work on dataflow machines [6] and its descendants such as the EARTH execution model [7]. A codelet is a group of machine instructions that are executed atomically. In an application written following the codelet model, all the code is partitioned into codelets. A codelet is fired (move to "ready for execution" state) once all the data and resources required become available. Codelets presuppose that all data and code required for execution is local. In this way, the need to explicitly program in the body of a codelet to hide latencies from accesses to remote data is eliminated.

The codelet model was developed to maximize performance and scalability. To achieve these goals, the programming interface to codelets is designed to enable the expression of different kinds of parallelism and give the control of how to express parallelism to users. Because of all the information needed to specify parallelism and synchronizations, it is often difficult to program directly in terms of codelets. For this reason, high-level abstractions built on top of codelets, which increase the programmability and readability without sacrificing performance and scalability, are of great importance.

In this paper, we propose using Hierarchically Tiled Arrays (HTAs) as a high-level abstraction for writing programs to execute on runtime systems based on the codelet model. HTAs [8] are a class of objects that encapsulate tiled arrays and data parallel operations on them. With HTAs, it is possible to describe most classes of parallel computations in a natural manner. The main motivation behind the design of HTAs is that, for a wide range of applications, tiling is universally used to control locality and parallelism. Our goal in the project described here is to show that HTAs can be efficiently mapped onto codelets without introducing global synchronization barriers, and that at the same time they hide the low level details of the runtime from the user. Note that our data parallel abstractions may not be the only mechanism for programming, they could coexist with other mechanisms including explicit invocation of codelets.

The remainder of the paper is organized as follows: Section II talks about the need for a high-level programming abstraction for the codelet execution model and proposes the HTA paradigm as a solution. Section III describes our assumptions of the baseline codelet model used in our design. Section IV explains how HTA programs can be mapped onto the codelet execution model. Section V discusses a few design decisions that can impact performance. Section VI provides an example demonstrating how parallel algorithms can be written with HTAs more easily. Section VII gives a brief description of the related work. Finally, Section VIII concludes.

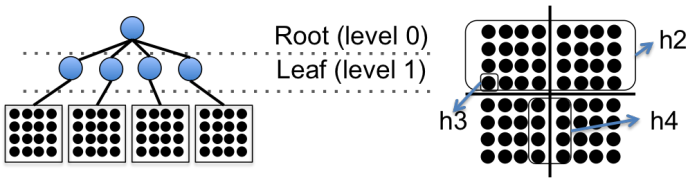

Fig. 1. HTA tree structure and accessing the contents of an HTA.

```
S0: Tiling tiling = new Tiling(new Tuple(2, 2),
S1:                            new Tuple(4, 4));
S2: HTA h1 = new HTA(2, tiling, HTA_DOUBLE);
S3: HTA h2 = h1(0, : );
S4: double h3 = h1(0, 1)[3, 0];
S5: double *h4 = h1[4:7, 3:4];
```

Fig. 2. An example of HTA construction and accessing elements.

II. Background

A. The Codelet Execution Model

The codelet execution model evolved from the dataflow execution model, in which the unit of computation is very fine-grained (often a single operation) and a program expressed in this model requires hardware support for efficient execution.

In contrast, in the codelet model, the units of computation, codelets, can be fine-grained computations but not as fine-grained as a single operation. A codelet program contains the definition of codelets and the dependences between codelets. A software runtime system, required for program execution, is responsible for keeping a record of dependences and scheduling codelets for execution when their dependences are satisfied.

Programmers do not have to explicitly specify the execution order of codelets or assign them to physical execution units. Instead, the runtime system decides when and where codelets are executed once their dependences are satisfied, so that the workload can be balanced dynamically. In addition, the codelet runtime system can make decisions to reduce energy consumption. One way is to provide mechanisms that proactively aim at reducing energy-consumption by applying techniques such as percolation [3]. The other way to reduce power consumption is to let the runtime system react to the behavior when some nodes exceed the power threshold by activating energy saving measures or migrating one or more codelets to a different compute node.

In order to get the most out of the codelet model, the existing applications have to be rewritten with the algorithms redesigned in terms of codelets. Programmers need to analyze the programs carefully and break down the computations into codelets and their data dependences (as opposed to synchronizations for control dependences) so that the parallelism can be exploited by the codelet runtime system. While it allows programs to be written in a way that retains great parallelism, these requirements result in the need for programmers to write low-level programs. This makes porting an existing application or developing a new one an onerous activity.

B. Hierarchically Tiled Arrays

We believe the HTA programming paradigm is a great solution to the programmability difficulties of codelets. HTAs have been successfully implemented in Matlab and C++, and have been studied for both distributed and shared memory environments [9], [10].

Essentially, an HTA program can be seen as a sequential program containing operations on tiled arrays. It has been demonstrated that HTA code is expressive, concise, and easy to reason about. It is also simple to start from a baseline sequential program and parallelize it with the HTA notations. The model provides a global view of data which lets tiles and scalar elements be easily accessed through chains of index tuples without explicitly specifying communication functions to fetch the data required. These features improve programmability and minimizes application development time.

Application codes written in the HTA notation are portable across different classes of machines since low-level details are not exposed to the users. For example, a map operation can be implemented using a parallel for loop, or it can be implemented as SPMD computations. Users write the same code using the high-level constructs provided by the HTA paradigm and they need not know the details of the underlying machines. Compared with completely rewriting existing applications using codelets, it can be preferable for application developers to use the more familiar programming paradigm provided by HTA while still enjoying the benefits of executing applications on codelet runtime systems. In the remainder of this section, we introduce the basic notations of the HTA programming model.

1) Data Representation: Regular arrays are flat, which means that they are composed of scalar elements. HTAs are hierarchically tiled so that an HTA tile can consist of not only scalar elements but also *smaller* HTAs. The data structure used to store hierarchical information is a tree (Fig. 1). At each level of the tree, a tuple associated with the tree node stores the tiling of the lower level. The root node is at level 0. Its children are at level 1, and so on. We use the term *leaf level* to refer to the tiles that are the lowest level tree nodes, and *scalar level* to refer to the scalars within the leaf level tiles.

The data in an HTA consists of the metadata and the raw data. The raw data contains the values used in the computation. The metadata includes the hierarchy, the distribution, and all the other information that are required for manipulation of HTAs. In the example shown in Fig. 1, the black circles are the raw data and everything else is the metadata.

2) Construction: There are several ways to create HTAs. One way is by specifying the hierarchy of tiles in the form of a sequence of N dimensional tuples. On distributed memory machines, the desired distribution of the tiles and the layout are also required arguments for the HTA constructor. An example program in C++ syntax is shown in Fig. 2. S0 creates an object `tiling` containing two tuples: 2×2 at the first level and 4×4 at the second level. It is then used in S2 to construct the two-level HTA object `h1`. The limitation of the method is that it can only create regular partitions—tiles of the same level have the same shapes with the same number of dimensions and same sizes along each dimension. Many flexible ways to create irregular HTAs are described in [8].

3) Accessing Elements: To access the elements of an HTA, two different access operators can be used: operator `()` for tile accesses, and operator `[]` for scalar element accesses. The operators accept a triplet of the form `low:high` or of the

form `low:step:high` for each dimension to indicate the range of selected elements. If either the `low` (or `high`) of the triplet is omitted, the corresponding minimum (or maximum) index is used. An extra `step` value can be given to access elements with strides other than 1.

By using access operators, programmers can construct new HTAs from the existing ones easily. For example, at S3 of Fig. 2, the operator `()` selects the 0^{th} row of tiles in `h1` to create a new HTA `h2` by copying data tiles from the row. S4 selects first the tile `(0, 1)` at level 1 in `h1`, and then `[3, 0]` selects a scalar element within the tile. S5 demonstrates the use of triplets, and using the operator `[]` at a non-leaf level tile results in a selection of a region across tile boundaries. In this case, the return value is a 2-D scalar array instead of an HTA. All the selected regions are illustrated in Fig. 1.

4) HTA Operations: These are either the high-level programming constructs or the API functions provided by the library for performing computations on HTAs conveniently.

Conformability is essential in HTA operations. In the HTA paradigm, conformability is a generalization of conformable arrays in Fortran90 to adapt the hierarchical and tiled nature of the data structure. We say two HTAs are conformable when they are of the same shape or when the smaller one of the two can be expanded to a shape that allows it to be operated on with the larger tile by tile. A scalar can be seen as an HTA of a single tile that has a single element, and it is always conformable to other HTAs. The detailed definition of conformability can be found in [11].

For some operations, the computations being performed on tiles of different HTAs do not require a strict ordering. These operations can be implemented using parallel programming constructs on a parallel machine. For example, in the construction of an HTA, the memory space for its data tiles can be allocated in parallel. On a sequential machine, it can still be executed in serial. The application codes are the same in both cases. The following list contains the classes of parallel HTA operations:

- Construction/destruction: The allocation and deallocation of memory space for HTA can be performed in parallel along with the initialization of data values within the tiles.
- Assignment Operation: The assignment operator = is used to modify the values stored in the left-hand side HTA. It performs conformability check before the actual assignment, and it adjusts the shape of the right-hand side HTA when needed.
- Arithmetic Operations: Basic arithmetic expressions (+, −, ⋆, /, negation, binary shift, ...etc.) are supported in HTA. The operations are pointwise operations (i.e. they are performed at the scalar level, element-by-element).
- Other Operations: Commonly used parallel operations are also supported. For example, `map(h, op, level)` allows applying an operator `op` at the tiles of the specified `level` of HTA `h`, and `reduce(h, op)` performs reduction on HTA `h` using the associative operation `op`. There are also operations that change the distribution of tiles or move data values around, including `circshift`, `transpose`, and `repmat`.

5) Program Execution: The execution of an HTA program can be divided into two parts. The sequential part includes everything that is not parallel operations. For example, the computation that updates the stack variables when calling a subroutine, forms the sequential part. The parallel operations, when executed, require multiple processes (or threads) working on different tiles of the HTA, possibly in parallel.

A trivial implementation would be to use the fork-join execution model, which uses one master process to execute the sequential part, and spawns slave processes whenever a parallel operation is encountered. The slave processes join back to the master when they are done with their assigned work, and the master process can proceed the execution of the sequential part until the next parallel operation happens. However, this approach would result in implicit barriers for all parallel operation invocations and hinders the execution performance.

The other option is to implement HTA in the SPMD execution model. The program execution starts with all available processes redundantly executing the sequential part. Since the sequential part is redundantly computed, all the processes have identical program state and thus a global view of data (i.e. they know how to acquire data tiles owned by the others). When an HTA operation is executed, a process examines whether the computation is owned (owner-computes: the owner of computation is the owner of the output data) by it. If so, the process gathers the input operands and executes the operation. Otherwise, it might still act as the producer and supply the data tiles to the consumer processes. When the parallel operation is completed, it can proceed to the subsequent sequential part of the program.

The major benefit of executing HTA code in SPMD fashion is that the implicit barriers are no longer required, since the owner of computation synchronizes with the producers of the input operands right before performing a parallel operation. This is exactly the minimal amount of synchronization needed due to actual data dependences. The execution can proceed once the data dependences are satisfied without having to wait for all other processes at a centralized synchronization point as opposed to the fork-join model.

III. The baseline codelet execution model

In the codelet execution model, program execution is described as a collection of non-preemptive units of computation—codelets, and their dependences. Dependences of a codelet are specified when it is created; they cannot be changed during the execution of a codelet. If some work requires changing dependences dynamically, it must be broken into a chain of codelets where each codelet in the chain depends on its predecessor and as well as on some extra other codelets. This continuation based computation is referred to as *split-phase* computation in [3]. We use "continuation" and "phase" interchangeably in the later description. Codelets are scheduled to execute when dependences are satisfied. Since the execution is dependence-driven without a linear program-order restriction, the parallelism can be exploited.

Several assumptions are made concerning the codelet execution model:

- Proximity-aware: Codelets are associated with data and are assigned to certain proximity groups. Codelets assigned to the same group can share data with each other quickly or even directly through shared memory accesses. In contrast, codelets assigned to different groups suffer a higher latency when they need to interact. Unless the user explicitly changes the group assignment, the runtime system has to ensure the *closeness* (in the sense of interaction speed) of codelets in a group. The idea is a generalization of the Threaded Procedure proposed in [3].
- Uniquely-identifiable: Each execution instance of the codelet has a unique identifier assigned to it by the user at creation time. To satisfy a dependence, only this identifier is needed. The operation satisfying the dependence does not need to know the location of the codelet. The feature can be supported on a distributed system efficiently if the underlying runtime system implements a distributed registry with mechanisms similar to dynamic hash table.
- Dependences are data dependences. Each dependence involves a pair of producer-consumer codelets and a data item. At the consumer's end, the dependence is one of the slots of dependences allocated at the codelet creation time. At the other end, the producer codelet satisfies the dependence with the identifier of the consumer, the slot index, and the dependent data item. The action causes a synchronization between the pair. The consumer codelet is fired when *all* slots are filled, and it can then acquire the dependent data items from the slots.
- Send-buffered, read-deferred: When a codelet satisfies a dependence, it is possible that the consumer codelet has not been created yet. A possible strategy is to buffer the data item associated with the dependence, since we do not wish to have the producer codelet blocked waiting. (This wait would be necessary if the codelet later modifies again the data item associated with the dependence. This situation would create an anti-dependence which might force the codelet to wait before the assignment until the data is consumed.) The runtime system rejects the send request by raising an exception when the system resource is not enough. On the other hand, when the consumer is ready to receive data, it is simply queued by the runtime system and will be fired when the dependences are satisfied.

In order to describe our design more clearly, several fundamental operations for the codelet model are defined here. We use *codelet* to refer to an execution instance and we explicitly say it when we refer to the code definition.

- `codelet_create(state, codelet_def, codelet_id, dep_count)`:
  The runtime system creates a codelet and associates to it a unique `codelet_id` specified by the user. The codelet is then queued and waits for the dependences to be satisfied. The runtime system scheduler is responsible for scheduling the execution of a codelet when all its incoming dependences are satisfied.
  A `codelet_def` is a function pointer referring to the code to be executed by the codelet. `dep_count` is the total number of dependence slots that need to be satisfied before the codelet can start execution. The parameter `state` represents the initial execution state of the codelet, which includes the stack, the register file, and the program counter. When `state` is `null`, the codelet executes the `codelet_def` from an empty initial state. In our approach, all codelets execute the same code and we use the operation differently by copying the program `state` and the `codelet_def` of the creator codelet so that the created one starts executing in the same code as the creator immediately after the `codelet_create` operation. Our usage is described in the next section. Semantically, it is similar to the UNIX `fork()` system call.
- `dep_satisfy(consumer_id, slot, item)`:
  The producer invokes this operation to satisfy a dependence slot of the consumer with a data item. `consumer_id` is the identifier of the consumer. It is assumed that the runtime system can efficiently locate and communicate with the consumer with the identifier. `slot` is the index number of the dependence slots of the consumer. `item` is the object encapsulating the dependent data.

IV. Mapping HTA onto codelets

In this section, we describe the design to map HTA programs to the codelet execution model. Let us first consider HTA creation which requires a distribution function D that, when invoked, returns the name of the owner codelet of the tile. The function is many-to-one. Tiles of either the same HTA or from different HTAs can belong to the same codelet and a single tile never belongs to two different codelets.

$$owner = D(HTA\_name, indices)$$

Tiles are distributed to the same owner usually because the computation of the owner require *fast* access to these tiles, so associating ownership implies binding the data tiles to a *locus* (location). Note that binding to a locus is conceptually different from assigning a hardware compute node since the number of different loci is not limited by physical resources.

We call *binder* a codelet that serves as the locus of the set of tiles and *root* a codelet that acts as the entry point of the program and is responsible for the creation and deletion of binder codelets. The root does not own any tiles.

We propose an approach that implements HTAs as a library on top of codelets (it is possible to enhance the support by building a compiler but that is beyond the scope of this work). An HTA program executes a sequence of statements, some of which operate on HTAs. The program can be conceived as sequential although it typically executes in parallel. Parallelism can be achieved in one of the following two ways:

1) The non-HTA part executes like any sequential program and the operations on HTAs execute in parallel. This form resembles a fork-join OpenMP program.

2) The program is executed in SPMD form. Each task owns a collection of tiles and executes the non-HTA part redundantly and only the part of the HTA operations that involves the tiles it owns. This is the mode we use in the implementation described in this paper.

Determinate program execution is guaranteed since an HTA program has sequential semantics although as we said the execution is typically in parallel.

To describe the work of the codelets, we only need to consider assignment statements (consisting of an expression at the right-hand side and a variable at the left-hand side), `HTA_create` and `HTA_destroy` statements, since the other HTA constructs can be converted to combinations of these statements.

```
00: HTA_create(name, numtiles, ownerlist, dist){
01:     /* build HTA metadata in program state */
02:   for (i = 0; i < numtiles; i++){
03:     owner = dist(name, i);
04:     if(isroot()){
05:       if(!exists(owner))
06:         codelet_create(mystate,mycode,owner,0);
07:     }
08:     if(isbinder()){ // one of the binders
09:       if(owner == myid)
10:         allocate_tile(name, i, mylocus);
11:       /* perform any other necessary actions */
12:     }
13:   }
14: }
```

Fig. 3. Implementation of `HTA_create` in terms of codelet operations.

Fig. 3 shows an exemplary implementation of `HTA_create`. The HTA program execution starts from the root codelet. Whenever the root finds `HTA_create`, it calls `codelet_create` (Line 06) if the binder codelet that owns the tile has not been created yet. The new binder codelet starts execution with a `state` that is identical to the execution state of the root right after the corresponding `codelet_create` statement (Line 07) and allocates the memory space required for the tiles locally in the same locus (Line 10). On the other hand, when the codelets who do not own the tiles (including the root) execute an `HTA_create` statement, they simply gather the information about the creation (Line 01) so they are "aware" of it.

When `HTA_Destroy` is executed, the binder codelets containing the tiles of the HTA have to deallocate the memory space for the tiles. All the other codelets who do not own any tiles of the HTA being destroyed also execute the function to remove the metadata from the program state. When a binder codelet does not own any living tiles anymore, it can be terminated to release system resources.

Assignment statements are handled differently depending on their types. For our strategy, there are four types of assignment statement:

1) Assignment statements involving no HTAs are executed sequentially by all codelets.
2) Assignment statements involving an HTA on the left-hand side (LHS) but not on the right-hand side (RHS) are neither executed by the root codelet nor the binder codelets that do not own any tiles referenced on the LHS of the statement. Any binder codelet containing at least one tile of the LHS HTA is the owner of computation and will carry out the assignment to the owned tiles directly, since the RHS expression can be evaluated locally.
3) An assignment statement containing HTA references on the RHS and non-HTA variable (either stack variable or local heap variable) on the LHS is an update to the program state. Since all codelets maintain identical program state, all codelets (including the root codelet) depend on the evaluation result of the RHS expression.
   In this case, all of the binder codelets containing referenced tiles are the *producers* and they have to satisfy all other codelets which are the *consumers*. To accomplish this, each codelet creates a continuation codelet associated with the same set of tiles and its `num_deps` equal to the number of owners of the referenced tiles. Finally, the continuation codelets start execution when the dependent tiles are received and evaluate the RHS expression and then store the result to update the program state.
4) Assignment statements involving HTAs on both sides is slightly more complex. In this case, the root codelet skips these statements. The binder codelets containing at least one tile any of referenced HTAs on the RHS are the *producers*. In contrast, the binder codelets containing at least one tile of the HTA on the LHS are the *consumers*.
   Each producer containing some tiles of the referenced HTAs on the RHS has to collect the sets of tiles required by the consumers and satisfies them correspondingly. Each consumer creates a continuation codelet, and the `num_deps` is equal to the number of producers who are responsible for supplying its dependent data. When the producer of some tile is the same as the consumer, as an optimization, the dependence can be implicitly satisfied.

Next we show some examples of the program execution. Before we start, our naming conventions for codelets are described. A name is associated with each codelet at creation time. The rules to generate names for binder codelets and the root codelet are as follows:

- The root codelet is named R with an superscript number indicating the continuation phase. For example, the first instance of the root is R^1, the continuation of it is R^2, and so on.
- Each binder codelet is named T concatenated with a number representing the index of the locus and a superscript number representing its phase. For example, $T5^3$ means the binder codelet who lives at the 5^{th} locus is at the 3^{rd} phase of execution.

Fig. 4 is an example program showing the HTA program execution, and the interactions between codelets during execution are shown in Fig. 5. The execution starts from the root codelet R^1. When it executes `HTA_create` at S2, it knows the HTA B is to be created with four tiles cyclicly distributed to two loci (0 and 1). At this point, there are no binder codelets

```
S0: double x, y, z;                          // R^1
S1: HTA B, G;
S2: HTA_create(B, 4, [0,1], CYCLIC);         // T0^1, T1^1
S3: HTA_create(G, 4, [0,1], CYCLIC);
S4: x = sin(y) * cos(z);
S5: B = x;
S6: z = reduce(B + G);                       // R^2, T0^2, T1^2
S7: G = B(-1: );                             // T0^3, T1^3
S8: B(0) = pow(B(1), 10);                    // T0^4
S9: G(2) = pow(G(3), 10);                    // T0^5
```

Fig. 4. An example of HTA program.

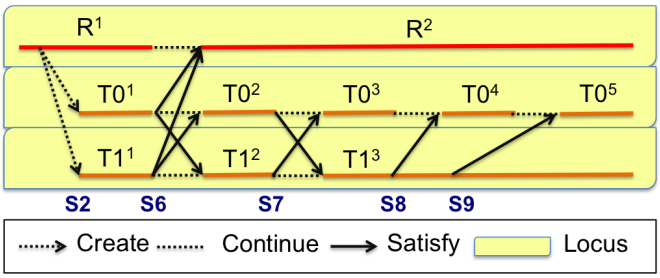

Fig. 5. The execution of codelets for the example in Fig. 4.

in the system. Thus, R^1 invokes `codelet_create` which creates $T0^1$ and $T1^1$, passes its current program state as input, and proceeds to S3, where R^1 does not create new binder codelets since tiles of HTA G are distributed to the existing binder codelets.

Now that $T0^1$ and $T1^1$ are created, their executions start at S2 and they allocate tiles of B locally. At S3, they both allocate tiles of G without having to interact. The next few lines demonstrate the different types of assignment statements. S4 is an assignment statement which is evaluated locally since no HTAs are involved (type-1). Next, S5 involves all tiles of B on the LHS, thus $T0^1$ and $T1^1$ both have to execute. $T0^1$ assigns to B(0) and B(2) and $T1^1$ assigns to B(1) and B(3), but no interaction is needed (type-2).

S6 is a type-3 assignment with an update to the stack variable living in the program state. Everyone including the root codelet has to enter a new phase, and the producers satisfy everyone with the tiles involved in the RHS expression. When the dependences are satisfied, $T0^2$ assigns to B(0) and B(2), and $T1^2$ assigns to B(1) and B(3).

At S7, the RHS is a permutation of tiles (circular shift) in B and the result is assigned to the tiles of G. The RHS and the LHS both involve HTA references, so it is a type-4 assignment. In this case, since R^2 does not own tiles, it omits the statement. $T0^2$ sends copies of B(0) and B(2) to $T1^3$, and $T1^2$ sends copies of B(1) and B(3) to $T0^3$. After the sending, $T0^2$ and $T1^2$ both enter new phases since they require new incoming dependences.

S8 and S9 are also type-4 assignments. They both select some tile owned by T1, and assign it to a tile owned by T0. The consumer T0 has to enter new phases in order to get new dependent data tiles, but the producer T1 stays at phase 3. Notice that a producer codelet may want to satisfy a consumer codelet at different phases. If we do not distinguish the different phases of the consumer using unique identifiers in `dep_satisfy`, when the satisfactions arrive out-of-order, the consumer may receive incorrect values.

From the example, we can see how the proposed approach can successfully map the HTA program execution to the codelet execution model. In the next section, we propose mechanisms to further extend and improve the design for execution performance.

V. Performance Considerations

The previously described approach to map HTA programs to the codelet execution model works, but we have not addressed much about its performance. Here we point out the possibilities for optimizing the execution performance of HTA programs on the codelet model.

A. Exploiting Finer-grain Parallelism

Letting each binder codelet perform all the computations associated with its owned tiles does not exploit the parallelism fully. For instance, in Fig. 5, S8 and S9 do not have to be executed in order since they work on different HTAs. But the current approach can not exploit this parallelism because the binder codelet sequentially performs all the work associated with its tiles.

Performance can be further improved if we exploit finer-grain parallelism of the computations owned by each binder codelet. One possibility is to let the binder codelet create a new type of *worker* codelet that performs part of the computations associated with the tiles owned by the binder. These workers execute in parallel with the binder, and do not perform the redundant computation required to maintain identical program state, since they only need the information necessary for the fine-grain computation and terminate when the computation is finished. The nested parallelism can be exploited in a way similar to the hybrid MPI and OpenMP programming paradigm.

B. Reuse and Locality

In the proposed approach, each binder codelet processes statements one-after-another dynamically. Thus, it cannot detect the reuses across multiple statements in the HTA program. One possibility for exploiting the locality across statements is to implement a cache of remote data tiles in the HTA library with some support from the underlying codelet runtime system. When an incoming dependence is deemed required for the continuation, the binder codelet looks up in its local cache for the tile of the specific revision (a revision counter associated to each tile would be necessary for this purpose). If it hits in the cache, the local cache replaces the remote binder who produces the tile to satisfy this particular dependence locally. On the other hand, a binder codelet producing data tiles works the same way as proposed, but the runtime system at its end performs a handshake with the consumer's end before actually sending data. During the handshake, the runtime at the consumer's end looks up its cache to see if the tile has been received before. If there is a hit, it notifies the producing end and cancel the communication.

VI. Example

LU decomposition is one of the fundamental methods for solving systems of linear equations. Block LU decomposition

```
S0: for k = 0 to n-1
S1:   lu(A(k, k))
S2:   A(k, k+1:) = mldivide(A(k, k).lt, A(k, k+1:))
S3:   A(k+1:, k) = mrdivide(A(k+1:, k), A(k, k).ut)
S4:   A(k+1:, k+1:) = A(k+1:, k+1:) - A(k+1:, k)
                      * A(k, k+1:)
```

Fig. 6. LU Decomposition in HTA notation.

Statement	Input	Output
$S1$	$IN(S1) = \{A_{kk}\}$	$OUT(S1) = \{A_{kk}\}$
$S2$	$IN(S2) = \{A_{kk}, A_{kj}\}$	$OUT(S2) = \{A_{kj}\}$
$S3$	$IN(S3) = \{A_{kk}, A_{ik}\}$	$OUT(S3) = \{A_{ik}\}$
$S4$	$IN(S4) = \{A_{ij}, A_{ik}, A_{kj}\}$	$OUT(S4) = \{A_{ij}\}$

Fig. 7. Sets used to calculate the dependence relationship for LU decomposition. k is a fixed value in each iteration of the for loop, and the values of i and j are replaced by the owned tile indices.

is the tiled version of it. The algorithm can be easily written in HTA notation as Fig. 6 shows. The matrix operations are implemented in HTAs with the same names as Matlab routines. The algorithm takes an HTA A as input which consists of $n \times n$ tiles, and performs the block LU decomposition in place.

In Fig. 6, S1 calls `lu` to perform a sequential LU decomposition on one tile A_{kk} in each iteration k. Next, S2 uses the HTA method `lt` to fetch the lower triangular part of A_{kk}, and performs `mldivide(A, B)` (which returns x for $Ax = B$) for all tiles in the kth row with column index $j > k$. Similarly, S3 invokes `ut` on A_{kk} to take the upper triangular part, and then uses it in the `mrdivide(A, B)` operations (which returns x for $xA = B$) with all tiles in the kth column with row index $i > k$. Finally, the last statement S4 can be seen as two forall loops with iteration space $k < i < n$ and $k < j < n$, and the innermost loop body performs $A_{ij} = A_{ij} - A_{ik} * A_{kj}$ to update the tiles in the submatrix.

Fig. 7 shows the sets used in dependence computation. Let the tiles of A be distributed to a 2D array of loci in a one-to-one mapping so that each tile A_{ij} is owned by the locus with index (i, j). In each iteration, k is a fixed value, and the values used for i and j in the IN and OUT sets are the locally owned tile indices (same as the locus indices in this mapping). Fig. 8 illustrates the dependence relationship of applying the block LU decomposition algorithm to a 3×3 block matrix at `k = 0`. Each binder codelet waits for only minimal data dependences before starting execution. For example, A_{11} can start the computation of S4 once A_{01} and A_{10} have been produced. No global barriers are required between statements.

Since the algorithm works on tiles with both column and row indices larger than the iteration count k, there is a load imbalance for the computations associated with tiles to the upper left are less than the ones to the lower right. The problem can be relieved by distributing the tiles cyclicly to the loci. This can be easily achieved by changing the distribution specified at HTA creation without having to modify the algorithm. When the loci are load-balanced, it is easier for the codelet runtime to map them to hardware computation resources.

If we attempt to implement the same algorithm directly using codelets, we will have to implement the distributed algorithm and orchestrate the creation and the dependences of codelets in the program manually. With an algorithm of this

Fig. 8. Example of block matrix for LU decomposition.

scale, programming codelets is still reasonably tamable, but it will definitely take more than the five lines of code used in the HTA notation. Consequently, it takes more time to write and is harder to maintain. While introducing reasonable overhead and redundancy, the high-level programming constructs provided in the HTA paradigm greatly reduce software development time.

VII. Related work

One of the most promising execution models that have begun to address the exascale challenges is the codelet model [3]–[5]. This model is inspired by previous works on dataflow models. A comprehensive survey on the classical dataflow execution models can be found in [12]. In particular, the codelet model incorporates some of the ideas and advantages of the macro-dataflow models [13], where the granularity is not defined at the instruction level but a coarser grain, and also of the hybrid dataflow/Von Neumman EARTH system [7]. Recently, several runtime systems have been proposed to exploit the codelet execution model, ETI's SWift Adaptive Runtime Machine (SWARM) [4], Delaware Adaptive Run-Time System (DARTS) [14], and the Open Community Runtime (OCR) [15].

In addition to the codelet model, we can find in the literature noticeable efforts to develop new efficient execution and programming models. A dataflow inspired programming and execution model was adopted as part of the TERAFLUX project [16]. In the TERAFLUX framework, OmpSs [17] tasks are translated to finer dataflow threads (DF-Threads) to be executed on a dataflow architecture. The codelet and DF-Threads models are very close in such a way that codelets can be mapped to DF-Threads. In particular, the authors ported DARTS [14] to the TERAFLUX infrastructure. The operations we described in Section III can also be implemented using DF-Threads. Another example is Intel Threading Building Blocks [18], which is a library implemented in C++ that allows expressing parallelism using high-level program constructs. The parallelism is obtained by defining tasks that can be performed concurrently, relying on the runtime system to split and map tasks to available hardware threads. Cilk [19] is an extension to the C language that uses a finer-grain execution model to take advantage of the asynchronous task creation for expressing parallelism. The Habanero execution model [20] also relies on expressing programs as a collection of asynchronous tasks. The Concurrent Collection (CnC) model [21] is a high-level programming model implemented upon Habanero, and it is inspired by dynamic dataflow.

The programming interface to codelets is often low-level since it provides maximum flexibility and fine-tuning opportunities. To deal with this problem, in this work we propose the use of the HTA library as a high-level abstraction for codelets. HTAs emphasize the concept of tiling both to express locality and parallelism. In the previous work, the advantages of using

HTAs were shown on shared and distributed memory multiprocessors [9], [10]. Several other similar libraries that provide a global view of the data structures exist, such as Global Arrays [22] and POET [23]. Both require SPMD programming style and explicit synchronizations which complicate programming. POOMA [24] also provides a global view of the data, but it lacks mechanisms to manipulate tiles easily and to decompose them hierarchically. From the languages side, we find PGAS (Partitioned Global Address Space) languages as Co-Array Fortran [25], UPC [26] and X10 [27], which offer a global view of the data as well as locality information to distinguish between remote and local accesses. These languages only provide information to the compiler about arrays to be tiled in a certain way in order to be distributed, but they do not allow to manipulate those tiles.

VIII. Conclusion

Our ongoing work is implementing the HTA library on two promising codelet runtime systems —ETI SWARM and OCR. We plan to evaluate the performance on x86_64 systems and explore more applications such as graph algorithms and larger-scale scientific applications.

The work we present in this paper is among the first attempts to build a high-level abstraction upon the codelet execution model. We propose a design which lets HTA programs be executed in terms of codelets efficiently without any global barriers, and it provides a good base for further exploration and research opportunities. We point out the features of the codelet runtime system that are required for us to realize the design, and we also suggest several possible components that can be implemented in the runtime system which will help to not only mitigate the difficulties in implementing the HTA library but also reduce the overhead and thus increase the execution performance. We discuss many design choices and their performance implications, which provide useful insights to runtime system and compiler developers, and application programmers whose work involves the codelet execution model.

Acknowledgment

This material is based upon work supported by the Department of Energy [Office of Science] under Award Numbers DE-SC0008716 and DE-SC0008717. This work has been also supported by the Xunta de Galicia (Spain) grant EM2013/041.

References

[1] S. Ashby et al., "The Opportunities and Challenges of Exascale Computing – Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee," 2010.

[2] P. Kogge et al., "ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems," 2008.

[3] G. R. Gao, S. Zuckerman, and J. Suetterlein, "Toward an execution model for extreme-scale systems – runnemede and beyond," CAPSL Tecnhical Memo 104, Department of Electrical and Computer Engineering, University of Delaware, May 2011.

[4] C. Lauderdale and R. Khan, "Towards a codelet-based runtime for exascale computing: Position paper," in *Proc. of the 2nd Int. Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, 2012, pp. 21–26.

[5] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao, "Using a "codelet" program execution model for exascale machines: Position paper," in *Proc. of the 1st Int. Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, 2011, pp. 64–69.

[6] J. B. Dennis, "First version of a data flow procedure language," in *Programming Symposium, LNCS*, 1974, pp. 362–376.

[7] K. B. Theobald, "EARTH: and effcient architecture for running threads," Ph.D. dissertation, McGill University, Montreal, Canada, 1999.

[8] B. Fraguela et al., "The Hierarchically Tiled Arrays programming approach," in *Proc. of the 7th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems (LCR)*, 2004, pp. 1–12.

[9] J. C. Brodman, B. B. Fraguela, M. J. Garzarn, and D. A. Padua, "Design issues in parallel array languages for shared memory," in *Proc. of the 8th Int. Workshop on Systems, Architectures, Modelling and Simulation*, ser. Lecture Notes in Computer Science, vol. 5114, 2008, pp. 208–217.

[10] J. Guo, G. Bikshandi, B. B. Fraguela, M. J. Garzaran, and D. Padua, "Programming with tiles," in *Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2008, pp. 111–122.

[11] G. Bikshandi, "Parallel programming with hierachically tiled arrays," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2007.

[12] W. M. Johnston, J. R. P. Hanna, and R. J. Millar, "Advances in dataflow programming languages," *ACM Computing Surveys*, vol. 36, no. 1, pp. 1–34, Mar. 2004.

[13] V. Sarkar and J. Hennessy, "Partitioning parallel programs for macro-dataflow," in *Proc. of the ACM Conference on LISP and Functional Programming*, 1986, pp. 202–211.

[14] J. Suettlerlein, S. Zuckerman, and G. R. Gao, "An implementation of the codelet model," in *Proc. of the 19th Int. Conf. on Parallel Processing (Euro-Par)*, 2013, pp. 633–644.

[15] "Open Community Runtime," https://01.org/open-community-runtime, accessed: 2014-06-30.

[16] R. Giorgi et al., "TERAFLUX: Harnessing dataflow in next generation teradevices," *Journal of Microprocessors and Microsystems*, April 2014.

[17] A. Duran et al., "OmpSs: a proposal for programming heterogeneous multi-core architectures." *Parallel Processing Letters*, vol. 21, no. 2, pp. 173–193, 2011.

[18] J. Reinders, *Intel Threading Building Blocks*, 1st ed. O'Reilly & Associates, Inc., 2007.

[19] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Proc. of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995, pp. 207–216.

[20] R. Barik et al., "The Habanero multicore software research project," in *Proc. of the 24th ACM SIGPLAN Conf. Companion on Object Oriented Programming Systems Languages and Applications*, 2009, pp. 735–736.

[21] Z. Budimlic et al., "Concurrent collections," *Scientific Programming*, vol. 18, no. 3-4, pp. 203–217, 2010.

[22] M. Krishnan, B. Palmer, A. Vishnu, S. Krishnamoorthy, J. Daily, and D. Chavarria, *The Global Arrays Users Manual*, 2012.

[23] R. Armstrong, "POET (Parallel Object-oriented Environment and Toolkit) and frameworks for scientific distributed computing," in *Proc. of the 30th Hawaii Int. Conf. on System Sciences: Software Technology and Architecture*, 1997, pp. 54–63.

[24] J. Reynders et al., "POOMA: A framework for scientific simulations on parallel architectures," in *Parallel Programming in C++*. MIT Press, 1998, pp. 547–588.

[25] R. W. Numrich and J. Reid, "Co-array fortran for parallel programming," *ACM SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, 1998.

[26] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren, "Introduction to UPC and language specification," CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

[27] P. Charles et al., "X10: An object-oriented approach to non-uniform cluster computing," in *Proc. of the ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications*, 2005, pp. 519–538.