Dataflow Execution of Hierarchically Tiled Arrays

Chih-Chieh Yang[1(✉)], Juan C. Pichel[2], and David A. Padua[3]

[1] IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598, USA
chih.chieh.yang@ibm.com

[2] CiTIUS, Universidade de Santiago de Compostela, Santiago de Compostela, Spain
juancarlos.pichel@usc.es

[3] University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
padua@illinois.edu

Abstract. As the parallelism in high-performance supercomputers continues to grow, new programming models become necessary to maintain programmer productivity at today's levels. Dataflow is a promising execution model because it can represent parallelism at different granularity levels and to dynamically adapt for efficient execution. The downside is the low-level programming interface inherent to dataflow. We present a strategy to translate programs written in Hierarchically Tiled Arrays (HTA) to the dataflow API of Open Community Runtime (OCR) system. The goal is to enable program development in a convenient notation and at the same time take advantage of the benefits of a dataflow runtime system. Using HTA produces more comprehensive codes than those written using the dataflow runtime programming interface. Moreover, the experiments show that, for applications with high asynchrony and sparse data dependences, our implementation delivers superior performance than OpenMP using parallel for loops.

Keywords: Parallel programming · Dataflow · High-level programming abstraction · Parallel algorithm

1 Introduction

Over the last decade, the pursuit of system performance has moved from increasing processor frequency to increasing the number of processing cores so that today's supercomputers can contain millions of cores [24]. This number is likely to increase significantly as we move to exascale systems. New notations will be

This material is based upon work supported by the Department of Energy [Office of Science] under awards DE-SC0008716 and DE-SC0008717, the National Science Foundation under award 1533912, MINECO under award RTI2018-093336-B-C21, the Xunta de Galicia under award ED431C 2018/19, and the European Regional Development Fund.

© Springer Nature Switzerland AG 2019
R. Yahyapour (Ed.): Euro-Par 2019, LNCS 11725, pp. 304–316, 2019.
https://doi.org/10.1007/978-3-030-29400-7_22

necessary for these future systems to keep the complexity of parallel programming at a manageable level. Such notations will rely on runtime systems [10] that create a simplified machine model and can better deal with applications whose performances depend on various dynamic decisions, such as scheduling and data movements.

Dataflow is a promising model for runtime systems. In a conceptually simple notation, it captures multiple levels of parallelism needed for efficient execution on exascale systems. It uses task graphs where tasks (i.e. sequential code segments) are represented by nodes and can be scheduled to execute as soon as their incoming data dependences (represented by the graph edges) are satisfied. Compared with conventional models which rely on programmers use of control dependences, a dataflow model utilizes inherent parallelism in programs naturally. Although implementations of such model [2,4,6,15–17,20] have shown great potential for exploiting parallelism, many of them lack high-level programming abstractions to attract programmers. To program using a native dataflow notation is a daunting task, because its programming style is unfamiliar and the learning curve is steep. Moreover, even when one learns to program in this way, the resulting codes could contain numerous dependence edges and lack structure, making these codes difficult to debug and maintain.

In this paper, we propose using Hierarchically Tiled Arrays (HTA) [1,5,11,12] as high-level abstractions to exploit the benefits provided by dataflow runtime systems, while helping productivity with a familiar programming interface for those trained to program in conventional notations. We implemented a fully functional HTA library on top of the dataflow-based Open Community Runtime (OCR) [8,17], and show through experiments that our design preserves the benefits of OCR while removing the need to program in the task graph notation.

The remainder is organized as follows. Section 2 gives an overview of the programming model HTA and OCR, the dataflow runtime system of our choice. Section 3 describes the design and implementation of HTA as a library on top of OCR (HTA-OCR). Section 4 presents the performance evaluation of our HTA implementation using various benchmark applications. The related work is described in Sect. 5. Finally, the conclusions are presented in Sect. 6.

2 Background

2.1 Overview of Hierarchically Tiled Arrays

An HTA program can be conceived as a sequential program containing operations on tiled arrays or sub-arrays. With HTA, programmers express parallel computations as tiled array operations. Because tiles are a first class object, the HTA notation facilitates the control of locality, which is of great importance today and will be even more so for future exascale systems.

By expressing computations in terms of high-level tiled array operations, programmers can focus on designing algorithms for maximal parallelism and better data locality and leave the mapping to the target machine and runtime system,

```
1 HTA A(2,              /*dimension*/
2       2,              /*levels*/
3       Tuple(N, N),    /*flat array size*/
4       Tuple(X, X));   /*tiling         */
5 HTA B(A.shape()), C(A.shape());
6 A.init(RANDOM); B.init(RANDOM); C.init(RANDOM);
7 for(k = 0; k < X; k++) {
8  for(i = 0; i < X; i++) {
9   C(i,{0:X-1}) += A(i,k) * B(k,{0:X-1}); }}
```

Listing 1. Example of tiled matrix-matrix multiplication in HTA.

including synchronization operations, to the HTA implementation which, for the one reported here, took the form of a library.

In HTA, programmers explicitly express parallelism by choosing the *tiling* (i.e. partitioning a flat array into tiles) of arrays. Multilevel tiling can be used, and each level can be tiled for different purposes. For example, there can be a top-level tiling for coarse-grain parallelism, a second-level tiling for fine-grain parallelism, and a third level for data locality in the cache.

An example of an HTA program in C++-like syntax is given in Listing 1. The code first constructs three HTAs, `A`, `B` and `C`. Each of them is an $N \times N$ array partitioned into $X \times X$ tiles. `A`, `B` and `C` are initialized, and then a two-level nested **`for`** loop performs tiled matrix-matrix multiplication. The parenthesis operator represent tile accesses and the curly braces represent array range selections. For instance, `C(i,{0:X-1})` selects the `i`th row of tiles in `C`. The operator `*` performs a matrix multiplication of two tiles. In this code segment, the creation, initialization, multiplication and addition assignment are HTA operations, and the control loops are sequential statements. There are plenty of parallelization opportunities, but the exact way is hidden from the user in the HTA library implementation of operations and memory access.

It has been shown [5] that HTA programs are expressive and concise. It is particularly convenient to parallelize an existing sequential program by replacing parallelizable computations such as for loops with operations on tiled arrays. Even without existing sequential code, using HTA facilitates building parallel applications from scratch. HTA programs are also more portable, since they are written in high-level abstractions without machine dependent details.

2.2 Overview of Open Community Runtime

Open Community Runtime (OCR) is a product of the X-Stack Traleika Glacier project [18] funded by Department of Energy of the US government. Its goal is to provide a task-based execution model for future exascale machines through software and hardware co-design.

In OCR, computations are represented as directed acyclic graphs where nodes are event driven tasks (EDTs, called simply *tasks* hereafter) that operate on relocatable data. The OCR API provides functions to create *objects* including *tasks*, *events*, and *data blocks*. Tasks represent computation, data blocks represent data

used or produced in the computation, and events are used to describe either data or control dependences between tasks.

The execution of OCR tasks is dictated by events. Tasks are not scheduled for execution immediately after their creation. Instead, at creation, an OCR task is placed in a queue, and the runtime system keeps track of its incoming dependences. When all the incoming dependences of an OCR task are satisfied, the task becomes *ready* and the runtime system can schedule it for execution. Tasks run to completion without ever being blocked, since all the data needed for the computation is available when they are scheduled.

Since task execution depends only on data blocks passed to tasks and not on data in the call stack or global heap objects, OCR runtime system can freely relocate tasks, as long as the data blocks needed can be accessed at the place of execution. The fact that both tasks and data blocks are relocatable makes it possible for the runtime system to make dynamic scheduling decisions for workload distribution, energy saving, and various other optimizations. This saves application programmers from having to optimize application code with machine specific details. However, to program directly using the OCR programming interface, one has to formulate computations as a dataflow task graph. It is a verbose way of programming since every task and dependence have to be explicitly specified. It is also difficult to maintain and debug code written in such fashion. In the next section, we explain how this weakness can be overcome by bridging the gap with HTA.

3 Design and Implementation of HTA-OCR

In this section, we describe the main ideas behind our HTA library implemented on top of OCR. We call this library HTA-OCR. Our goal is to take advantage of both the programmability of HTA and the performance benefits of OCR's dependence-driven execution. Interested readers can find more details in [26].

3.1 Program Execution

An HTA-OCR program starts with a master task which executes the program sequentially except for HTA operations that are typically executed in parallel. The library routine implementing an HTA operation analyzes the operands (HTA tiles) and determines the data dependences (if any) of the subtasks performing operation. Then, the routine invokes OCR routines to create the subtasks and specifies their dependences in the form of OCR events. If it does not depend on the results of the subtasks, the master task then continues executing subsequent statements of the HTA program without waiting for the subtasks to complete, possibly overlapping execution with the subtasks.

Figure 1 shows a two-statement code segment and its dataflow task execution. The program operates on three 1-D HTAs, each containing tiles. The first statement assigns the content of `B` tiles to the corresponding tiles of `A`. The second performs another assignment from the tiles of `A` to those of `C`. Obviously, there

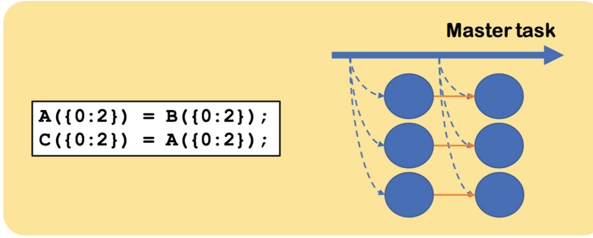


Fig. 1. HTA-OCR task graphs of assignment operations. The thick blue arrow represents the master task execution; The blue circles represent subtasks; The dotted blue thin arrows represent subtask creations; The orange thin arrows represent the data dependences between tasks. (Color figure online)

are flow dependences between the two statements. The right-hand side of the figure, shows the dynamically constructed task graph. The master task is represented as a thick blue arrow to show that its execution typically spans longer than other subtasks, represented as circles. The dotted blue thin arrows represent subtask creations, and the orange thin arrows represent the data dependences between tasks. When the master task executes the first statement, three subtasks are spawned and each copies one tile of `B` to the corresponding tile of `A`. Next, three more subtasks are spawned for the second statement. The second group of subtasks have incoming data dependences (the continuous arrows) from the first group due to the flow dependences on the tiles of `A`.

OCR shared memory implementation manages multiple worker threads upon which tasks can be scheduled. As soon as the incoming dependences of a subtask are satisfied, it is ready and can be scheduled for execution. The worker threads use a work-stealing scheduling algorithm so that tasks can be stolen by other threads for load balancing. To best utilize computing resources, the program should generate abundant subtasks that have sparse dependence edges among them, so that there is a higher probability of having numerous ready tasks to be scheduled at any given time.

3.2 Data Dependences

In the execution style described above, parallel tasks spawn dynamically with their dependences discovered by the master task examining and updating access record of HTA tiles. *Non-HTA variables* of global scope are always assigned in the master task. On the other hand, *HTAs* are only assigned in subtasks. Considering the two types of variables, data dependences exist whenever, in program order, a location is first written and then read or written, or is first read and then written. Four different cases exist:

1. The assignment to an HTA tile depends on some non-HTA variable accessed previously by the master task.
2. The assignment to a non-HTA variable depends on some non-HTA variable accessed previously by the master task.

3. The assignment to a non-HTA variable depends on some HTA accessed by a previous HTA operation.
4. The assignment to an HTA tile depends on some HTA accessed by a previous HTA operation.

In Case 1, the data dependence is resolved automatically, since the master task would have evaluated the non-HTA variable at the spawning point of the subtask to compute the HTA. Thus, the variable can be passed by value into the subtask. Similarly, in Case 2, the data dependence is guaranteed to be resolved because non-HTA variables are always evaluated by the master task in program order. In Case 3, the non-HTA variable assignment has data dependence on the completion of subtasks. Since in OCR, block-waiting for tasks is not supported, we implemented a split-phase continuation mechanism, explained in 3.3. In Case 4, the new subtasks must wait for the results of previous subtasks, and therefore data dependence arcs must be created. For this case, we developed a tile-based dependence tracking mechanism which utilizes access record of HTA tiles to ensure that tasks using the same HTAs always access them in the correct order respecting the data dependences.

3.3 Split-Phase Continuation

During program execution, the master task sometimes must wait for the results of a subtask. When it discovers incoming dependences from subtasks, it creates a new *continuation* task which is a clone of itself, and passes the original program context (including the program stack, the register file, and the program counter) to the clone, along with a list of new dependence events. As a result, the runtime system will only schedule the continuation task after both the original master task completes and the new incoming data dependences are satisfied. When it starts, it restores the program context and then continues the execution with new inputs.

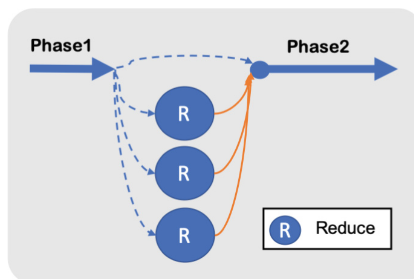


Fig. 2. Split-phase continuation. The thick blue arrow representing the master task execution which splits into two phases due to the new dependences on the reduction results. The continuation resumes executing the main program when it receives the reduction results. (Color figure online)

Consider a reduction on a 1-D HTA of three tiles, illustrated in Fig. 2. The master task (Phase 1) spawns three subtasks and each sequentially performs reduction on its assigned tile to get a single scalar value. Since the number of subtasks is small, the three scalar values are sent directly to the continuation task (Phase 2) to be reduced to the final result. If the number of leaf tiles are large, a parallel tree reduction can be used.

4 Experiments

Table 1. List of benchmarks.

Source	Benchmark
Hand-coded	Cholesky
Teraflux [9]	Sparse LU
NAS parallel benchmarks [3]	EP, IS, LU, FT, MG, CG

We evaluate our HTA-OCR shared memory implementation using several benchmarks listed in Table 1. The baseline are OpenMP versions of these programs that use parallel loops *as the only parallel construct*. We use this baseline in two ways. First, for the first two benchmarks in Table 1, the baseline implementation helps us assess the benefit of overlapping subtasks from different vector assignments which are implemented as separate parallel loops in the OpenMP versions. Since OpenMP parallel loops execute a barrier at the end, this overlap is not possible in OpenMP and therefore the overlap is the reason for performance advantage of the HTA version. Second, for the NAS Parallel Benchmarks, the OpenMP implementation helps us evaluate the efficiency of our implementation. Because in the NAS Parallel Benchmarks the overlapping of subtasks from different array operations is very limited, and therefore differences in performance between the HTA-OCR and the OpenMP versions is efficiency of the implementation. The experiments are on a single-node with four Intel Xeon E7-4860 processors, each has ten cores. We use up to forty worker threads so that each thread binds to a dedicated core without hyperthreading. Our purpose is to compare the execution models, so we timed major computation and excluded initial setup.

4.1 Tiled Dense Cholesky Factorization

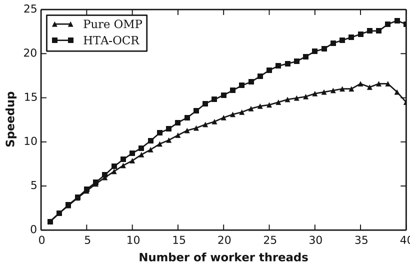
Cholesky factorization takes as input a Hermitian positive-definite matrix and decomposes it into a lower triangular matrix and its conjugate transpose. We use a tiled Cholesky fan-out algorithm. Intel MKL sequential kernels are used for tile-by-tile multiplication and tile triangular decomposition. The OpenMP version (Listing 2) factorizes a diagonal tile (Line 3), and uses the result to update the tiles of the same column in the lower triangular matrix (Line 5–6). The submatrix tiles in the lower triangular matrix are then updated using the results of the

```
1  for(int k = 0; k < n; k++) {
2     int numGEMMS = (n-k)*(n-k-1)/2;
3     POTRF(&A[k*n+k]);
4     #pragma omp parallel for schedule(dynamic, 1)
5     for(int i = k+1; i < n; i++)
6         TRSM(&A[i*n+k], &A[k*n+k]);
7     #pragma omp parallel for schedule(dynamic, 1)
8     for(int x = 0; x < numGEMMS; x++) {
9         int i, j;
10        GET_I_J(x, k+1, n, &i, &j);
11        if(i == j) SYRK(&A[j*n+j], &A[j*n+k]);
12        else       GEMM(&A[i*n+j], &A[i*n+k], &A[j*n+k]);}}
```

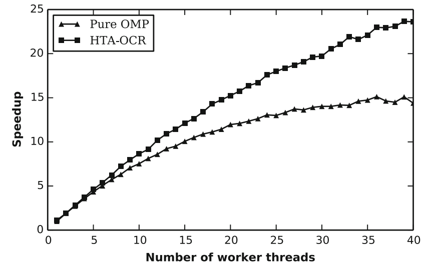
Listing 2. Tiled dense Cholesky factorization in OpenMP.

```
1  for(int k = 0; k < n; k++) {
2     map(POTRF, A(k, k));
3     map(TRSM, A({k+1:n-1}, k), A(k, k));
4     for(int j = k+1; j < n; j++) {
5        map(SYRK, A(j, j), A(j, k));
6        map(GEMM, A({j+1:n-1},j),A({j+1:n-1},k),A(j,k));}}
```

Listing 3. Tiled dense Cholesky factorization in HTA-OCR.



(a) 3200×3200 matrix, tile size 200×200



(b) 6400×6400 matrix, tile size 400×400

Fig. 3. Tiled dense Cholesky factorization results.

column tiles as input (Line 9–12). The updates in different parallel for loops have an implicit barrier in-between, and a **k** loop iteration blocks before all updates of the previous loop iteration complete. The HTA-OCR version (Listing 3) has a similar program structure, but it replaces the parallel for loops with the `map()` operations and array range selection, resulting in more concise code. During execution, the master task discovers dependences and constructs task graphs dynamically. No implicit global synchronization barriers are necessary.

In Fig. 3, we use two different problem sizes with the same partition of 16×16 tiles and plot the speedup over sequential execution under various number of worker threads. In both settings, the task granularity is large, and the curves

are similar. The HTA-OCR version has better speedup overall and scales better. It is about 1.5× faster at higher thread counts. As mentioned above, its advantage comes from eliminating the barriers which allows not only the tasks in the two different inner loops but also in the different outermost loop iterations to overlap. In contrast, implicit global synchronization barriers in the OpenMP version prevent task executions from overlapping even when required input data is ready, resulting in lower compute resource utilization.

4.2 Tiled Sparse LU Factorization

LU factorization converts a matrix A into the product of a lower triangular matrix L and an upper triangular matrix U. Adapted from the sparse LU code in Teraflux project [9], Listing 4 shows the HTA-OCR implementation. An outer `k` loop contains four steps:

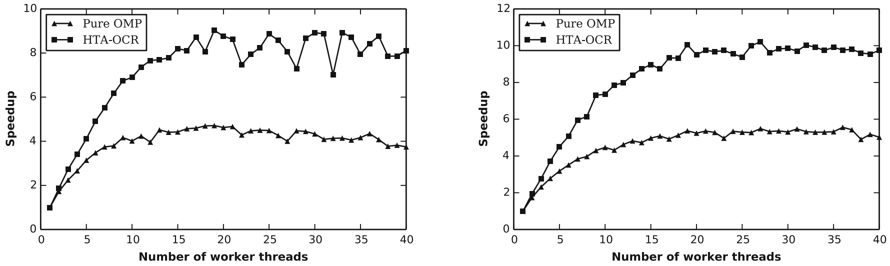
1. At Line 2, `DIAG` factors the diagonal tile `A(k,k)` into the lower triangular part `A(k,k).lt` and the upper triangular part `A(k,k).ut`.
2. At Line 3, `ROW_UPDATE` solves `X` for the equation `A(k,j)=A(k,k).lt*X` for `j = k+1 to n-1`.
3. At Line 4, `COL_UPDATE` solves `X` for for the equation `A(i,k)=X*A(k,k).ut` for `i = k+1 to n-1`.
4. At Line 5–8, `SM_UPDATE` updates each tile in the submatrix `A(i,j)-=A(i,k)*A(k,j)` if neither of `A(i,k)` and `A(k,j)` is all-zero.

```
1 for (k=0; k<n; k++) {
2  map(DIAG,A(k,k));
3  map(ROW_UPDATE,A(k,{k+1:n-1}),A(k,k));
4  map(COL_UPDATE,A({k+1:n-1},k),A(k,k));
5  for(i=k+1; i<n; i++)
6   if (A(i,k) != NULL)
7    for (j=k+1; j<n; j++)
8     if (A(k,j)!= NULL) map(SM_UPDATE,A(i,j),A(i,k),A(k,j));}
```

Listing 4. Parallel tiled sparse LU factorization in HTA-OCR.

The operations within a step are fully independent, but data dependences exist between different steps. There are also dependences across iterations of the `k` loop. The HTA-OCR library can dynamically construct a dataflow task graph by discovering the data dependences without user explicitly stating the dependences. Compared with Cholesky factorization in Sect. 4.1, the computation graph of LU factorization can be more complex, but the sparseness eliminates some nodes and dependence edges that would exist for the dense case. The OpenMP version uses a parallel for loop for each step and relies on implicit global barriers for the correctness.

The results of two problem sizes are shown in Fig. 4, both with 16 × 16 tiles and tile sizes are 100 × 100 and 200 × 200 respectively. Similar to the results



(a) 1600 × 1600 matrix, tile size 100 × 100 (b) 3200 × 3200 matrix, tile size 200 × 200

Fig. 4. Tiled sparse LU factorization results.

of tiled Cholesky factorization in Sect. 4.1, the HTA-OCR version shows greater scalability. It is close to 2× faster under forty threads. The advantages come from having no global barriers that may over-restrict task overlapping, just as in Cholesky factorization.

4.3 NAS Parallel Benchmarks

NAS (Numerical Aerodynamic Simulation) Parallel Benchmarks [3] are created by NASA for evaluating the performance of parallel supercomputers. We implemented six of them in HTA-OCR and observed the strong scaling results of class C as shown in Fig. 5. We plot the ratio of the HTA-OCR execution time over the OpenMP counterpart under the same number of threads. Most of them have a workload consists of regular computations that can be evenly divided easily and use global synchronizations. Because there is little opportunity for overlapping the execution of subtasks from different HTA statements, the main difference in performance between the HTA-OCR implementations and their OpenMP counterparts is overhead of execution. As can be seen in Fig. 5, in practically all cases there is less than 20% difference in performance and in some cases the HTA-OCR version is faster. We conclude that the performance of our experimental HTA-OCR implementation is competitive with that of the mature (an likely highly optimized) OpenMP library [13].

4.4 Summary of Experiments

For dense Cholesky factorization and sparse LU factorization, HTA-OCR shows superior performance than OpenMP. While HTA-OCR program complexity is similar to OpenMP, the dataflow runtime system can utilize CPUs effectively for the abundant asynchronous subtasks and their sparse data dependences. In contrast, OpenMP implicit barriers restrict task overlapping and this results in bad performance. Note that, if OpenMP Tasking is used instead of parallel loops, it

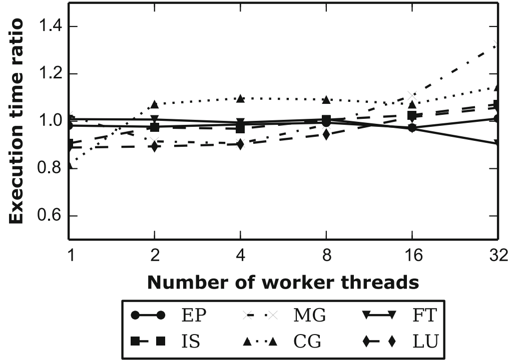


Fig. 5. NAS Parallel Benchmark results (Class C).

is possible to express the dataflow graph and execute in a data driven fashion. It would have comparable performance as HTA-OCR, but its code would be more cumbersome, since explicit data dependence annotations are needed. For the NAS Parallel Benchmarks, HTA-OCR shows decent results close to OpenMP. In most cases, the performance difference is within 20%. The HTA-OCR performance does not always surpass OpenMP, because the benchmarks mostly have easily-balanced workloads and bulk-synchronous execution which are ideal for OpenMP.

In all of our experiments, we present strong scaling results. The task management overhead (spawning, event satisfaction, scheduling, ..., etc) in OCR is significantly larger than the that of parallel loops in OpenMP. This makes HTA-OCR performance more sensitive to task granularity. To achieve good parallel efficiency, the task granularity has to be large enough to amortize the overhead. However, in strong scaling, as we use more threads, we partition a fixed-sized problem into more tiles and thus increasing task management overhead while decreasing task granularity. Devoting more future efforts into improving task overhead is crucial for the success of the dataflow runtime systems.

5 Related Work

OCR is based on the codelet model [14,27]. This model incorporates some of the ideas and advantages of the macro-dataflow models [23], where the granularity is defined not at the instruction level but a coarser grain one, and also of the hybrid dataflow/Von Neumman EARTH system [25]. Similarly, PaRSEC [6] is another runtime system that adopts dataflow model for coarse-grain task executions.

The Concurrent Collection (CnC) model [7] is a high-level programming model implemented upon both Habanero [21] and Intel Threading Building Blocks [22], and it is inspired by dynamic dataflow.

Charm++ [15] is a programming paradigm that also adapts the dataflow execution model for its runtime system design. Charm++ provides an object

oriented programming interface, thus it might be more suitable for application programmers to develop large parallel applications than OCR.

Based on the encouraging results of SMPSs [19], OpenMP Tasking was extended to support data dependent tasks. The expressible data dependences are limited to tasks within the same parallel section. In comparison, HTA-OCR tasks are not confined within parallel sections.

Legion [4] lets user write programs by decomposing application data into *logical regions* and explicitly spawning asynchronous tasks that operate on the regions. A software out-of-order processor dynamically infer data dependences. In terms of programming abstraction, Legion is lower-level than HTA, as parallel tasks are implicitly created in HTA.

6 Conclusions

This paper presents the design and implementation of the HTA programming model for execution on top of a dataflow runtime. Our work is among the first attempts to provide high-level programming abstractions upon dataflow runtime systems. We propose a strategy to map HTA programs onto dataflow task graphs, and we implemented the design as a fully functional HTA-OCR library whose important mechanisms were also discussed in detail. While our work describes data dependences in parallel programs among array tiles, we believe that our strategy can be extended to support other data structures, such as parallel sets, to provide a general-purpose programming paradigm. For performance evaluation, a variety of benchmarks were implemented using the HTA-OCR API and the experiments were conducted. The results show great promises of using HTA as programming abstractions upon dataflow runtime systems for its programmability and its ability to preserve the benefits from dataflow execution.

References

1. Andrade, D., Fraguela, B.B., Brodman, J., Padua, D.: Task-parallel versus data-parallel library-based programming in multicore systems. In: 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, pp. 101–110 (2009)
2. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. Concurr. Comput. Pract. Exp. **23**(2), 187–198 (2011)
3. Bailey, D., et al.: The NAS parallel benchmarks. Int. J. High Perform. Comput. Appl. **5**(3), 63–73 (1991)
4. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: expressing locality and independence with logical regions. In: International Conference on High Performance Computing, Networking, Storage and Analysis, p. 66 (2012)
5. Bikshandi, G., et al.: Programming for parallelism and locality with hierarchically tiled arrays. In: 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 48–57 (2006)
6. Bosilca, G., et al.: Parsec: exploiting heterogeneity to enhance scalability. Comput. Sci. Eng. **15**(6), 36–45 (2013)

7. Budimlic, Z., et al.: Concurrent collections. Sci. Prog. **18**(3–4), 203–217 (2010)
8. Budimlic, Z., et al.: Characterizing application execution using the open community runtime. In: International Workshop on Runtime Systems for Extreme Scale Programming Models and Architectures, in conjunction with SC15 (2015)
9. Consortium, T.: Teraflux applications (2017). https://svn.teraflux.eu/svnpub/apps/, Accessed 04 June 2017
10. Da Costa, G., et al.: Exascale machines require new programming paradigms and runtimes. Supercomput. front. Innovations **2**(2), 6–27 (2015)
11. Fraguela, B., et al.: The hierarchically tiled arrays programming approach. In: 7th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems, pp. 1–12 (2004)
12. Fraguela, B., et al.: Optimization techniques for efficient HTA programs. Parallel Comput. **38**(9), 465–484 (2012)
13. Free Software Foundation: Gomp - an openmp implementation for GCC. https://www.gnu.org/software/gcc/projects/gomp/, Accessed 01 Feb 2019
14. Gao, G.R., Zuckerman, S., Suetterlein, J.: Toward an execution model for extreme-scale systems - runnemede and beyond, May 2011
15. Kale, L.V., Krishnan, S.: Charm++: a portable concurrent object oriented system based on c++. In: Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications, pp. 91–108 (1993)
16. Lauderdale, C., et al.: Swarm: A Unified Framework for Parallel-for, Task Dataflow, and Distributed Graph Traversal. ET International Inc., Newark (2013)
17. Mattson, T., et al.: The open community runtime: a runtime system for extreme scale computing. In: High Performance Extreme Computing Conference, pp. 1–7 (2016)
18. Modelado Foundation: Traleika glacier project (2018). https://wiki.modelado.org/Traleika_Glacier, Accessed 01 Oct 2018
19. Perez, J.M., Badia, R.M., Labarta, J.: A dependency-aware task-based programming environment for multi-core architectures. In: International Conference on Cluster Computing, pp. 142–151 (2008)
20. Planas, J., Badia, R.M., Ayguadé, E., Labarta, J.: Hierarchical task-based programming with starss. Int. J. High Perf. Comput. Appl. **23**(3), 284–299 (2009)
21. Barik, R., et al.: The Habanero multicore software research project. In: Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications. pp. 735–736 (2009)
22. Reinders, J.: Intel Threading Building Blocks, 1st edn. O'Reilly & Associates Inc, Sebastopol (2007)
23. Sarkar, V., Hennessy, J.: Partitioning parallel programs for macro-dataflow. In: ACM Conference on LISP and Functional Programming, pp. 202–211 (1986)
24. Strohmaier, E., Dongarra, J., Simon, H., Meuer, M.: Top500 list (2008). https://www.top500.org/, Accessed 01 Oct 2018
25. Theobald, K.B.: EARTH: and effcient architecture for running threads. Ph.D. thesis, McGill University, Montreal, Canada (1999)
26. Yang, C.C.: Hierarchically Tiled Arrays as High-Level Programming Abstractions for Dataflow Runtime Systems. Ph.D. thesis, University of Illinois at Urbana-Champaign (2017)
27. Zuckerman, S., Suetterlein, J., Knauerhase, R., Gao, G.R.: Using a codelet program execution model for exascale machines: position paper. In: 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era, pp. 64–69 (2011)