© 2017 by Chih-Chieh Yang

HIERARCHICALLY TILED ARRAYS AS HIGH-LEVEL PROGRAMMING ABSTRACTIONS FOR DATAFLOW RUNTIME SYSTEMS

BY

CHIH-CHIEH YANG

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Doctoral Committee:

Professor David Padua, Chair, Director of Research
Professor William Gropp
Professor Laxmikant Kale
Dr. José Moreira, IBM Research
Associate Professor Juan Carlos Pichel, University of Santiago de Compostela, Spain

Abstract

In the foreseeable future, high-performance supercomputers will continue to evolve in the direction of attempting to build distributed, immensely parallel and highly heterogeneous machines. It is well known that in order to utilize these machines, good parallel programs are essential. However, conventional parallel programming models were created when supercomputers were smaller and more homogeneous. It is not clear whether these models will enable the same level of productivity for the next generation supercomputers. It is expected that intermediate runtime systems between software applications and the underlying hardware machine architecture will help abstract away the extreme complexity of future large-scale machines. In the recent past, there have been growing interests in dataflow execution models due to their flexibility in making dynamic decisions. Despite their advantages, the dataflow runtime systems tend to have a low-level programming interface that is difficult to tame. It requires the programmer to decompose the computation and write program to construct dependence graph explicitly, resulting in programs that are difficult to build, debug and maintain.

In this thesis, we repurpose the Hierarchically Tiled Array (HTA) programming model for improving the programmability of the dataflow runtime systems. HTA facilitates parallel programming by letting the programmer express algorithms as tiled array operations which contains implicit parallelism. We propose a design to map an HTA program to a dataflow task dependence graph dynamically, so that the programmer can write conventional HTA programs while enjoying the benefits provided by the underlying dataflow runtime system.

As a proof of concepts, we implemented our design for the shared memory environment and implemented a variety of benchmarks for performance evaluation. We found that, for applications with high asynchrony and sparse data dependences, our implementation results in simpler programs than those obtained by using the dataflow runtime programming interface and delivers superior performance results than OpenMP using parallel for loops. We also learned about the scalability issues in our current design and propose solutions as possible future work.

For my parents.

Acknowledgments

The Ph.D. program is a transformative experience which is very challenging not just intellectually but also mentally. I would like to thank my advisor David Padua for his guidance over the years. He always encourages me to question everything, think bigger, dig deeper, and never settle until our solutions are great. I also want to express my gratitude for my Ph.D. committee for their time and feedback on my thesis. Among them, Professor Juan Carlos Pichel also collaborated with me on this work, and I would like to acknowledge his ideas and insights to help improve my work.

Studying abroad for the Ph.D. program means leaving my family duty on hold for an extended period of time. It is not always smooth sailing, and I cannot thank my family enough for their love and support for me to pursue this achievement during these years. I wish that I have made them proud.

Table of Contents

List of Figures . . . . **ix**

List of Tables . . . . **xi**

List of Listings . . . . **xii**

Chapter 1 Introduction **1**

Chapter 2 Overview of Hierarchically Tiled Arrays **5**
- 2.1 Program Structure 7
  - 2.1.1 Data Types 7
  - 2.1.2 Array Operations 8
- 2.2 Execution Model 11
  - 2.2.1 Fork-join 12
  - 2.2.2 SPMD 13
  - 2.2.3 Dataflow Tasks 14

Chapter 3 Design of HTA-OCR **16**
- 3.1 Overview of Open Community Runtime 16
  - 3.1.1 Dependences in OCR 17
  - 3.1.2 OCR Objects 18
  - 3.1.3 OCR Execution Model 21
- 3.2 Mapping HTA Program to OCR Tasks 22

Chapter 4 Implementation of HTA-OCR **25**
- 4.1 Program Execution 25
- 4.2 Data Dependences 28
- 4.3 Tile-based Dependence Tracking 30
- 4.4 Implementation of Core HTA Operations 35
  - 4.4.1 HTA_map 36
  - 4.4.2 HTA_full_reduce 37
- 4.5 Split-phase Continuation 38
- 4.6 Prioritized Tasks 41
- 4.7 Tracing API 42

Chapter 5 Performance Evaluation **45**
5.1 2-D Convolution 46
5.2 Tiled Matrix-Matrix Multiplication 50
5.3 Tiled Dense Cholesky Factorization 56
5.4 Tiled Sparse LU Factorization 61
5.5 AMG Microkernel 66
5.5.1 Matvec Kernel 66
5.5.2 Relax Kernel 67
5.6 Parallel Mergesort 69
5.7 K-means 72
5.8 Parallel Breadth First Search 75
5.9 LULESH 78
5.10 NAS Parallel Benchmarks 83
5.10.1 EP 83
5.10.2 IS 84
5.10.3 FT 85
5.10.4 MG 87
5.10.5 CG 89
5.10.6 LU 90
5.11 Summary 92

Chapter 6 Towards A Scalable Design **94**
6.1 Overhead Analysis 94
6.2 A Scalable Design 97
6.3 Performance Optimizations 103
6.3.1 Support for Parallel Reductions 103
6.3.2 Curtailing Proxy Events 104
6.3.3 Graph Reduction 105

Chapter 7 Related Work **107**
7.1 SWARM 107
7.2 Charm++ 108
7.3 Legion 109
7.4 OpenMP Tasking with Data Dependent Tasks 109
7.5 Tensorflow 110
7.6 PaRSEC 111

Chapter 8 Future Work **112**

Chapter 9 Conclusions **115**

Appendix A Tiled Matrix-matrix Multiplication Full Programs **117**

Appendix B Tiled Cholesky Factorization Full Programs **124**

Bibliography **135**

List of Figures

2.1 HTA data structure 7
2.2 HTA with 8×8 scalar elements 9

3.1 Dependence graph formed by OCR objects 17
3.2 HTA-OCR program execution 23

4.1 HTA-OCR library code in comparison with user program 26
4.2 HTA-OCR mergesort program execution 28
4.3 HTA-OCR tile state machine 32
4.4 Subgraphs generated for HTA-OCR parallel operations 33
4.5 HTA-OCR task graph built with tile-based dependence tracking 34
4.6 Split-phase continuation 40
4.7 Program trace displayed by SWARM trace_viewer application 44

5.1 Single output element computation of convolution 2D with filter size 3×3 . 46
5.2 2-D Convolution results, 2048×2048 elements 49
5.3 Task graph of tiled matrix-matrix multiplication of 3×3 tiles 50
5.4 Tiled matrix multiplication results using naive kernel 54
5.5 Tiled matrix multiplication results using MKL `dgemm` kernel 55
5.6 Task graph of tiled Cholesky factorization of 4×4 tiles 57
5.7 Dense Cholesky factorization results 60
5.8 Sparse LU task graph with input HTA of 4×4 tiles 63
5.9 Sparse LU factorization results 65
5.10 Matvec kernel results 67
5.11 Relax kernel results 69
5.12 Parallel mergesort results 72
5.13 Kmeans results 74
5.14 Parallel BFS results 77
5.15 LULESH execution trace of a single timestep using 32 worker threads and edgeElem = 90 81
5.16 LULESH results 82
5.17 EP benchmark result 84
5.18 IS benchmark result 86
5.19 FT benchmark result 87
5.20 MG benchmark result 88

5.21 HTA-OCR MG V-cycle execution trace 89
5.22 CG benchmark result 90
5.23 LU benchmark result 92

6.1 The effects of task generation overhead 95
6.2 SPMD + Dataflow execution with multiple master tasks 98
6.3 Illustrations of the task synchronization in SPMD+Dataflow for the given program 102
6.4 Optimization to curtail the number of proxy events at the HTA library level 105

List of Tables

4.1 HTA-OCR tile states. 31
4.2 HTA-OCR tile state transition table 32
4.3 HTA-OCR task priority preset values. 42

5.1 PAPI hardware event counter report of AMG Microkernel Relax kernel execution 68
5.2 The configurations used in LULESH experiments 79

6.1 The values of the arguments to the hash function to generate unique GUID . 100

Listings

2.1 HTA creation . . . 9
4.1 HTA-OCR mergesort . . . 27
4.2 HTA-OCR program that requires split-phase continuation . . . 39
4.3 An operator function with tracing calls . . . 43
5.1 Convolution 2D in HTA-OCR . . . 47
5.2 Convolution 2D in OpenMP . . . 47
5.3 Tiled matrix-matrix multiplication in HTA-OCR . . . 50
5.4 Tiled matrix-matrix multiplication in OpenMP . . . 51
5.5 Tiled dense Cholesky factorization in HTA-OCR . . . 56
5.6 Tiled dense Cholesky factorization in OpenMP . . . 58
5.7 Code to generate parallel tiled sparse LU sparsity pattern . . . 61
5.8 Parallel tiled sparse LU factorization in HTA-OCR . . . 62
5.9 Parallel mergesort in HTA-OCR . . . 70
5.10 Parallel mergesort in OpenMP . . . 71
5.11 LULESH OpenMP code using parallel for loop . . . 79
5.12 LULESH HTA-OCR code using HTA_map() . . . 79
A.1 Pure OCR tiled matrix-matrix multiplication implementation . . . 117
A.2 HTA-OCR tiled matrix-matrix multiplication implementation . . . 120
A.3 Pure OpenMP tiled matrix-matrix multiplication implementation . . . 122
B.1 Pure OCR tiled Cholesky factorization implementation . . . 124
B.2 HTA-OCR tiled Cholesky factorization implementation . . . 129
B.3 Pure OpenMP tiled Cholesky factorization implementation . . . 132

Chapter 1

Introduction

Over the last decade, the pursuit of computer system performance has moved from increasing processor frequency to increasing the amount of processing cores in a system. While the physical limitations of the semiconductor process make Moore's Law ineffective for the reduction of cycle time, it can still be used for scaling up the processor core count, and in this way continue to deliver better performance in a cost-effective manner. Nowadays, parallel processors are prevalent in all kinds of computing. On one hand, in personal computing, processors in desktop computer, laptop computers and mobile devices commonly have 2 to 4 cores. On the other hand, in high-performance computing, large scale systems typically contain up to 10 million cores [42]. With this massive parallelism, the computing power of large-scale supercomputers has reached petaflops.

Nonetheless, just by having massive parallelism in the hardware architecture design is not enough for applications to run fast. To utilize parallel machines well, programmers usually need to scrutinize sequential program codes and manually convert them into parallel codes with a suitable parallel programming model for their machines. For general purpose processors, there are many options of parallel programming models to choose from. As much as some computer scientists wish to find one parallel programming model to rule them all,

there does not seem to be a supreme one that is the best choice in all cases.

In the past decade, two parallel programming models continue to stand out: OpenMP [8] and MPI [22]. OpenMP is known for its simplicity. By annotating sequential programs with compiler directives, programmers can adapt their existing code to get moderate speedup in a short time. Yet, there are a few shortcomings making it not the ideal choice for developing parallel applications. First, it is only supported in shared memory systems. Plus, OpenMP programs often heavily rely on the use of global synchronization barriers. These problems limit its performance scalability, since large-scale computing systems are often distributed memory systems where the global synchronization overhead is high. In distributed memory systems, MPI is the dominant parallel programming model due to its great performance scalability. MPI programs run in SPMD fashion, and multiple processes synchronize with each other when necessary. Compared with OpenMP, MPI is more difficult to learn and the complexity of MPI programs is usually higher. But MPI offers more flexibility for designing complex performance-optimal parallel algorithms. MPI also has great interoperability, and it is often used in a hybrid way (MPI+X) with other parallel programming models, including OpenMP. Both models are still evolving with new standard specifications coming out every few years.

However, going into the next generation of exascale computing, it is not clear whether these models will continue to perform well while keeping the complexity of programming at a manageable level. Adapting existing proven models for future computing generations is a popular topic, and there has been some studies on this topic [5, 21, 23]. As an alternative, researchers are also exploring new designs for a software stack that are suitable for future large-scale hardware architecture through a software-hardware co-design [38, 9]. A rising consensus in the field is that having a runtime layer [15] between applications and the hardware can reduce the complexity of application programs by presenting a simplified abstract machine model so that the user programs do not deal with superfluous details. A sophis-

ticated runtime system can also cope better with applications which need various dynamic decisions, such as scheduling and data movements, to be made dynamically at runtime for performance reasons.

Various runtime systems for parallel programs have been proposed in recent years. Among them, the ones inspired by dataflow philosophies are promising [36, 6, 33, 30, 27, 10]. Once a popular research area, the dataflow execution model has the potential to express the maximal amount of parallelism in a program by allowing the tasks represented by nodes in a dataflow graph to execute as soon as their input data dependences are satisfied. While earlier attempts to make fine-grained dataflow machines were not successful due to the lack of locality and the overhead of communication and synchronization, later research works [39, 44] suggest using code segments as nodes in the dataflow graph instead of the single operations used in the original dataflow systems. Coarsening the unit of execution amortizes the overhead in creating and scheduling nodes in a dataflow graph, resulting in a more efficient execution. Several terms have been used to describe similar ideas, such as the codelet program execution model [44] and macro dataflow [39]. In this thesis, we use a system called Open Community Runtime (OCR) [11, 32, 33]. Henceforth, the term *dataflow runtime system* will be used to refer to the runtime systems that uses the dataflow execution model.

Although some implementations of dataflow runtime systems [36, 30, 32, 3] have shown great potential for exploiting parallelism, many of them lack high-level programming abstractions to entice application developers. To write programs directly with the low-level programming interface provided by them is a daunting task, because the dataflow programming style is unfamiliar to most programmers and the learning curve is steep. Even when one learns to program in this way, the resulting code lacks a comprehensive structure, making it difficult to debug and maintain. The success of any new programming model depends largely on the acceptance of the software community. When there is no compelling reason, it is difficult to convince users to learn a new programming model from scratch. In this thesis,

we propose using Hierarchically Tiled Arrays (HTA) as high-level abstractions to exploit the benefits provided by dataflow runtime systems, while keeping the programming interface familiar to users who are trained to write in conventional notations so that the productivity can be ensured. An implementation of HTA is done to show that our design preserves the benefits provided by the underlying dataflow runtime system. The major contributions of our work includes:

1. A strategy to map imperative HTA programs onto dataflow execution.
2. A significant improvement of the productivity in software development on top of the dataflow runtime programming interface.
3. An implementation as the proof of concepts and an evaluation of the implementation with a variety of benchmarks. Performance analysis of the results is also conducted to discover issues for future improvements.
4. A tracing library that captures application execution traces for performance debugging and analysis.

The thesis is organized as follows. Chapter 2 gives an overview of the HTA programming model. Its program structure, the operations, and the execution model are described. Chapter 3 describes how HTA program can be executed on the dataflow runtime system, specifically for OCR. An overview of OCR is given first and then our design is explained. Chapter 4 provides the details of the HTA-OCR implementation and many of the important mechanisms are discussed. Chapter 5 presents the results of our experiments with various benchmark applications. In Chapter 6, we propose changes that could be applied for making the existing design scale up to larger systems. In Chapter 7, the related work is described, and the future extension to this work is described in Chapter 8. Finally, the conclusions are drawn in Chapter 9.

Chapter 2

Overview of Hierarchically Tiled Arrays

Hierarchically Tiled Array (HTA) [7, 19, 18, 2] was proposed as a programming model to simplify parallel programming. With HTA, programmers express parallel computations in terms of tiled array operations. They are also encouraged to design algorithms with data locality in mind, which is a very important principle while writing code for modern-day machine architecture that has a low-latency limited-sized hardware cache and high-latency memory accesses. The model has been implemented on top of MPI and it has been demonstrated to work efficiently. In this chapter, we give an overview of the HTA programming model. More details of our new implementation can be found in Chapter 4.

An HTA program can be seen as a sequential program containing operations on tiled arrays. Tiled arrays are logical storage spaces for data used in parallel algorithms, and operations on them can be parallelized in various ways. The optimal parallelization often depends on the underlying machine architecture or the runtime layer implementation. 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 low-

level machine-dependent parallelization details, such as synchronization between processors, to the HTA library implementation or the HTA compiler.

Programmers explicitly express parallelism by choosing the tiling of the array. 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 hardware cache.

The synchronization between parallel processors or threads is implicit. While some tiled array operations can be executed fully independently in parallel, the operations such as reductions require synchronizations. An HTA library implementation or a compiler can determine the need to synchronize among parallel processors or threads and hide it from the programmer.

It has been shown that HTA programs are expressive, concise and comprehensive. It is particularly convenient to parallelize an existing sequential program using HTA by replacing parallelizable computations such as parallel for loops with one or more operations on tiled arrays. Even without existing sequential code, using HTA can shorten the time to develop a fully working parallel application. Application developers can really boost their productivity when they develop parallel programs in HTA.

HTA programs are also more portable, since they are written in high-level abstractions without machine dependent details. For example, a map operation, which applies a function to each tile of an array, can either be implemented using a parallel for loop or as an SPMD computation. Users invoking a map operation in an HTA program can expect it to be parallelized either by the compiler or the library properly on different classes of machines. Although for optimization reasons, it is best for programmers to know the machine and parallelize programs accordingly using low-level parallelization mechanisms, with a sophisticated HTA implementation, it may still be desirable to trade off slightly sub-optimal performance for the greatly increased productivity and the reduced hassle of tailoring codes

for different machines.

2.1 Program Structure

In an HTA program, programmers use regular C primitive types, C structs, and HTAs as variables to formulate their algorithms. The operations involving HTAs are parallelizable array operations, and operations consisting only non-HTA variables are executed sequentially.

2.1.1 Data Types

Array is one of the most important data structure used in almost all non-trivial programs. While regular arrays usually contain data elements in some contiguous memory storage space, HTAs allow defining logical hierarchical tiling for multidimensional index space using tree representations. At the leaf level, besides the raw data elements, the tile metadata contains information for the dimension of the tile, the data type of the scalar elements, the pointer to the raw data, ... etc. At higher levels, a tile can contain elements which are also HTA objects so that a tree hierarchy can be formed. The tree hierarchy can be used to infer the data locality in the program. For example, in a cluster, an HTA can have coarse-grain, topmost tiles distributed to different NUMA nodes, and finer-grain tiles allocated in different NUMA domains of a single node.

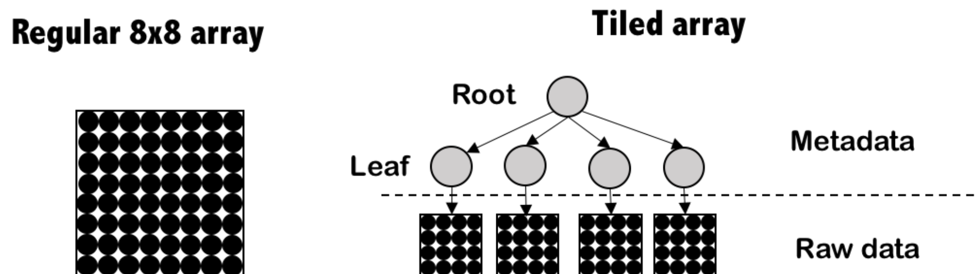


Figure 2.1: HTA data structure

An example of the tree representation of an HTA is shown at Figure 2.1. The left-hand side figure shows a regular two-dimensional array where the black dots are the scalar elements in the array. The right-hand side figure shows a two-level HTA containing 4 tiles arranged as a 2×2 mesh, and 4×4 scalar elements inside each tile. Both arrays in the figure have the same number of raw data elements, but the HTA representation requires extra storage space for metadata at different levels to provide descriptions of its tiling to facilitate tiled array operations.

A few other data types are needed in HTA. `Tuple` is a sequence of integers which is used in many different HTA operations. It can be used to specify the dimensions and the tiling of an HTA. It can also be used to index an element in an HTA. `Distribution` contains the information of how tiles are distributed in a system. `Region` specifies a subset of elements in an HTA for applying operations to partial data. `Partition` is used to reshape an existing HTA.

2.1.2 Array Operations

There are several categories of HTA operations including construction, destruction, accessor, assignment, pointwise, and collective operations. Most of them have inherent parallelism of various degrees. For example, consider assigning an HTA to another one of the same shape. Here, all assignment of tiles can be performed simultaneously. Higher-order operations such as reduction and prefix scan are also provided. These are implemented using well-known efficient parallel algorithms.

When tiled array operations are performed by parallel tasks, the size of leaf tiles determines the task granularity. It is the responsibility of programmers to partition the index space of an array into some suitable granularity to maximize parallelism and keep the parallelization overhead low. In contrast, the parallelization of the tiled array operations for each underlying parallel machine or abstract machine model presented by the runtime layer is

implemented by the HTA library. With compiler support, static analysis can be performed by the compiler to adjust task granularity by splitting or coarsening tasks.

Construction/Destruction

An HTA is constructed by calling `HTA_create()` function specifying the dimensions, the number of levels, the type of the scalar elements, the distribution of tiles, the layout, the dimensions of the untiled array, and the tiling. The library routine allocates memory space to store the raw data and extra space for metadata that describes the hierarchy.

```
1    Dist  dist = Dist_create(BLOCK, mesh);
2    Tuple flat = Tuple_create(2, 8, 8);
3    Tuple t0   = Tuple_create(2, 2, 2);   // 2x2
4    Tuple t1   = Tuple_create(2, 2, 1);   // 2x1
5    HTA* A = HTA_create(2, 3, flat, ORDER_TILE, dist,
        HTA_SCALAR_TYPE_DOUBLE, 2, t0, t1);
```

Listing 2.1: HTA creation

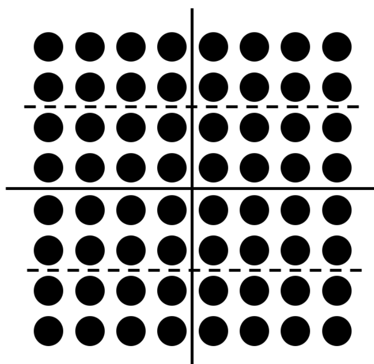


Figure 2.2: HTA with 8×8 scalar elements

In Listing 2.1[1], a 2-D three-level HTA is constructed. The `Tuple flat` specifies that the untiled array is 2-D and has 8×8 scalar elements. The tiling is formed by two `Tuples`, `t0` and `t1`. It specifies 2×2 tiles beneath the root level, and 2×1 tiles in the leaf level, resulting in leaf tiles of 2×4. The layout `ORDER_TILE` means that the scalar elements in a tile is stored

[1]The code here is C since the HTA-OCR implementation used in this thesis is a C library instead of C++. The reason for using C is because at the beginning of the OCR project, there was no plan for it to support C++. The HTA program would look less verbose if C++ was supported.

in a contiguous memory space. The `BLOCK` distribution makes the HTA be distributed as blocks of consecutive tiles onto a virtual processor mesh. Finally, `HTA_SCALAR_TYPE_DOUBLE` indicates that the data type of the scalar elements is double-point floating number.

The destruction of an HTA is simply done by invoking `HTA_destroy()`. Notice that the HTA specification does not require the memory allocation and deallocation to happen synchronously. Thus, in the case of HTA construction, it is possible for the library routine to return as soon as the HTA metadata is created and the actual allocation of the memory space for the raw data can be deferred to a later time, as late as the first attempt to use it. The same idea applies to HTA destruction.

Accessor Operations

Accessor operations are used to access an element of an HTA. It could either be an access to a tile or a scalar element, depending on the level being accessed. To use the accessor operation, programmers need to create a `Tuple` to specify the indices of the desired element. Chaining multiple `Tuple`s allows accessing deeper into the HTA hierarchy. In the C++ HTA implementation, the `[]` and `()` operators are overloaded for accessor operations. In the C implementation, since operator overloading is not a language feature, accessor operations are performed through functions such as `HTA_pick_one_tile()` which retrieves a handle to a lower-level tile specified by the tile indices (represented by a `Tuple`), `HTA_flat_read()` which reads a scalar element at a location specified by the global element indices, and `HTA_flat_write()` that writes an input value to a scalar element.

Pointwise Operations and Assignment Operations

Pointwise operations are applied scalar-by-scalar. They are fully independent and can be easily parallelized. Primitive operations including addition, subtraction, multiplication, and division are supported. Under this category, HTA also provides support for unary operation,

binary operation and assignments. For binary operation and assignments, there can be cases when the shape of the operand HTAs are different. In such cases, [7] specifies conformabiliity rules for checking the validity of the operations.

Collective Operations and Higher-order Operations

In HTA, collective operations are used to refer to those which change the structure of HTAs but not the individual scalar values. For example, a transposition changes the position of array elements, and permutation and shifting reorder array elements. While restructuring of HTAs has algorithmic purposes, it implies communication on distributed systems and requires multiple nodes to collectively perform the operation.

There are three higher-order operations in HTA: map, reduce and scan. Programmers can use custom operators in these operations and choose the tree level to apply them either on tiles or on scalars. Map applies a function to all elements of an HTA at a given level without any constraint on the order of individual operations performed on the elements. It is equivalent to using a parallel for loop that iterates over tiles or scalars at the specified level. Reduce and scan can also be applied to a specific level. The library implements efficient parallelization strategies for them depending on the underlying machine environment.

2.2 Execution Model

Since the HTA API only specifies the data representation and the operations that can be applied at a higher level, the actual program execution and how operations are parallelized are implementation dependent. For example, in cases when an HTA program is executed on a single core processor, it can be executed using a single thread with the operations on tiles executed in some sequential order. On a multithreaded shared memory machine with low overhead global synchronizations supported by the hardware, it can be executed in fork-join

fashion, similar to OpenMP parallel for loops. On a distributed memory machine, it can execute in SPMD fashion, where multiple processes execute the same HTA program. Upon encountering HTA operations, they follow owner-compute rule to perform partial computations based on the tiles they own, and synchronize using point-to-point communications or collective communications when necessary. Non-HTA data is replicated and operations involving this type of data are executed by all processes redundantly. Earlier, we proposed [43] a mapping of HTA program to the dataflow execution model using SPMD execution. In this thesis, we investigate a new way of executing HTA programs on dataflow runtime systems. In the following sections, we discuss the three possibilities.

2.2.1 Fork-join

It is possible for an HTA program to be parallelized in fork-join fashion if we build the HTA library on top of parallel programming models such as OpenMP or Intel Thread Building Blocks [37]. In such cases, a master thread executes an HTA program starting from the program entry point. When a parallelizable tiled array operation is encountered, the library routine performs the operation using parallel constructs provided by the underlying model. An implicit barrier happens at the end of each array operation, hence it is called "fork-join".

An issue with this approach is that for each array operation there is overhead in starting the work of the parallel threads. The other issue is that application performance can suffer from having frequent global barriers at the joining point. This causes overhead and possible load balancing problems. Thus, if the tiling used in the array operations does not result in balanced computation workload, computing resources could be wasted waiting for the thread that takes the longest to complete its parallel subtask. The parallel efficiency can only be good when the workloads for all of the array operations are balanced well. This is often not a realistic assumption for practical applications. Even if the programmer balances the workload for certain machines, the performance is not guaranteed to be good if the code is

executed on other machines.

2.2.2 SPMD

The original HTA work implemented in MPI uses the SPMD execution model. In this version, the sequential part of an HTA program is redundantly executed by a fixed, user-specified number of *processes*. During array construction, each *process* locally builds the whole hierarchical tree metadata structure of the HTA while owning only a subset of the raw data tiles. The distribution of the data tiles to processes is specified by the user at creation. Metadata records the ownership of all data tiles so that each process has a global view of the tiled arrays since it has enough information to locate remote tiles.

The HTA operations are executed by processes following the owner-computes rule. When a process executes an HTA operation, it knows the partial computation which it is responsible for. It determines the data dependence of the operation and performs point-to-point communications to either supply or acquire data tiles. It then performs the partial computation in parallel to other processes. In this mode, a parallel process is stalled only if the data tiles it depend on are remote and have not all been received yet. If the workload is not balanced, a process can still finish its own partial computation and move on before other processes, similar to the semantic of OpenMP `nowait` clause.

However, in this model, there are a few problems that could limit application performance. First, the choice of the number of processes P poses an upper bound to the exploitable parallelism. Even if there is a large amount of parallelism and the user creates enough tiles to expose the parallelism, some of the subtasks are serialized and only P-fold parallelism is exploited.

Second, in order to acquire data from another process, the processes involved need to perform two-way synchronization. For example, a process could be well ahead of others in the program execution when it executes a parallelizable array operation. Suppose it must

supply data to or receive data from a remote processes, it then needs to block and wait for the other to reach the same operation and performs the corresponding two-way synchronization before it can continue. The amount of asynchronous execution between processes is limited.

Finally, due to the previous issue, the workload for each tiled array operation needs to be balanced by distributing tiles to processes carefully. But it is not always possible to balance workload by regular data decomposition. For example, in tiled Cholesky factorization algorithm, if we factor the lower left triangular matrix, the tiles to the lower right have much larger workload than the tiles to the upper left.

2.2.3 Dataflow Tasks

In recent years, dataflow runtime systems have gained popularity due to their potential usefulness for large-scale high performance computers. In this thesis, we explore the possibility to execute HTA programs by dynamically converting them into a dataflow task graph.

The unit of work in dataflow runtime systems are data dependent tasks. Programs are typically dynamically-constructed task dependence graphs where the nodes are tasks and the edges represent data dependences. The tasks are asynchronously created and their dependences explicitly specified by the programmer at creation time. They stay in a pool after creation, and the dataflow runtime system tracks whether their input data dependences are satisfied. When the dependences are satisfied, the tasks become ready and the runtime system schedules them for execution on available computing resources. The major advantages of using dataflow runtime systems are its flexibility in making dynamic decisions and its ability to maximize parallelism, since there is no strict execution order enforced by programmers on the tasks.

In such model, we can use a single task (we refer to it as the *master* task hereafter) to execute a HTA program from its entry point. To exploit parallelism in array operations, we can let the master task spawn a set of new subtasks whenever it encounters an array oper-

ation to perform it collaboratively. The master task needs to specify the data dependences controlling the execution of the new subtasks at their creation time, so that the runtime system can take over the responsibility of monitoring the data dependences and fire the subtasks when the inputs are ready. Ideally, the master task spends minimal time in creating subtasks asynchronously, without having to wait for the completions of them.

The runtime system maintains a pool of all tasks created, and schedule them for execution on available computing resources when their data dependences are satisfied. Since tasks do not consume computing cycles when their inputs are not ready, the computing cycles are spent mostly on meaningful computation instead of spin waiting. Naturally, this allows higher parallel efficiency when there is enough parallelism exposed to the runtime system. When the subtasks complete, they notify the runtime system that their outputs are generated, so the subtasks depending on their outputs can start. In Chapter 3, we go into more details on mapping an HTA program execution to a dataflow task graph, specifically for the OCR runtime system.

Chapter 3

Design of HTA-OCR

This chapter describes how an HTA program can be mapped to a dataflow task graph for executing on a dataflow runtime system. Specifically, we use Open Community Runtime (OCR) in our work. In Section 3.1, an overview of OCR is given to help readers understand the underlying runtime system. Section 3.2 explains how an HTA program is converted to an OCR task graph so that it can be executed by the OCR runtime.

3.1 Overview of Open Community Runtime

Open Community Runtime [11, 33] is a product of the X-Stack Traleika Glacier project [17] funded by Department of Energy. Its goal is to provide a task-based execution model for future exascale machines through software and hardware co-design. The major contributors of the OCR specification are the researchers at Intel and Rice University. The OCR reference implementation was officially released in May, 2015 [32].

The philosophy of OCR is to form computations as directed acyclic graphs (DAGs) where nodes are event driven tasks (EDTs, referred to as 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. All OCR objects are associated with unique identifiers (GUIDs) at creation time and they are globally addressable, which means that a GUID can be used to refer to an object in any OCR function without distinguishing whether the object is local or remote.

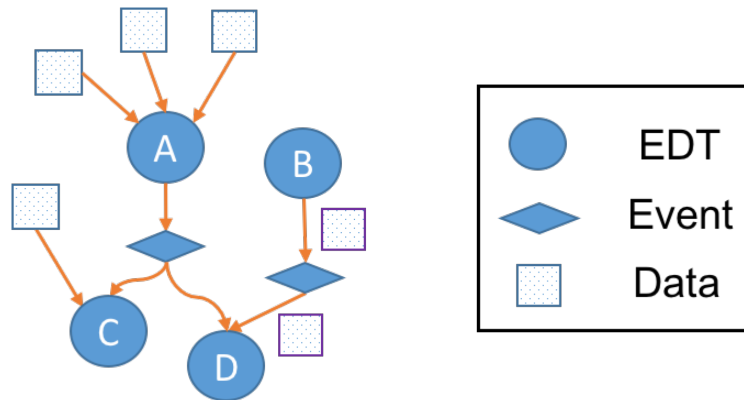


Figure 3.1: Dependence graph formed by OCR objects

Figure 3.1 shows an OCR task graph. The blue circles represent tasks, the blue diamonds are events, the squares are data blocks and the orange arrows represent dependences. In this example, Task A has three input dependences from three different data blocks. Task B does not have input dependences and it is ready for execution right after it is created. Task A satisfies an output event, which is connected to Task C and D, and the links do not carry data and thus they represent control dependences. In contrast, the indirect link that connects B and D through an event carries the purple data block produced by Task B.

3.1.1 Dependences in OCR

The dependences in OCR are represented as links between objects. Links are directional and they are defined when programmers explicitly set them to connect from a source end-point to a destination end-point. In OCR, source end-points are called *post-slots*, and destination

end-points are called *pre-slots*.

OCR objects usually have a single post-slot. For a task, the post-slot is satisfied when the task completes execution. For a data block, it is immediately satisfied when the object is created. For an event, its post-slot is satisfied when the event is triggered. Notice that it is not required for a post-slot to be connected to a pre-slot of another object.

Tasks and events can have multiple pre-slots. The pre-slots are the input dependences and need to be explicitly connected to with program code. A task can be scheduled for execution only when all of its pre-slots are satisfied.

With links between pre-slots and post-slots, both data dependences and control dependences can be expressed. When a link source end-point (post-slot) is satisfied with a data block, the object at the destination end-point (pre-slot) can receive the data block. This exemplifies a data dependence. If the program code satisfies a post-slot without a data block, it represents a control dependence.

3.1.2 OCR Objects

Here we describe the three major objects: tasks, events, and data blocks. We do not cover all the details and only present the concepts necessary for understanding the thesis. If the reader wishes to know more, please consult the OCR documentation [32].

Event Driven Tasks

EDTs (tasks for short) are units of work in OCR applications. They can be scheduled for execution by the OCR runtime system only after their input dependences are satisfied. They are non-blocking, meaning that when they start executing, they proceed till the end and cannot be blocked by the actions of other OCR objects.

To create a task using the OCR API, the programmer has to define an *EDT function* which contains the work that needs to be done. The function pointer is fed to `ocrEdtCreate()`

along with a list of 64-bit values, and a list of dependences (i.e. the pre-slots of the task). The call to `ocrEdtCreate()` returns two GUIDs. One is the GUID of the task, and the other signifies the post-slot which is satisfied when the task completes execution and can be connected to other OCR objects.

It is possible to insert `ocrEventSatisfy()` or `ocrAddDependence()` inside of an EDT function body. This can be seen as adding post-slots to the task object so that the links from the task object do not all come from the same source and not carry the same data.

Events

OCR events are used for setting up links between OCR objects. An event has a single post-slot, and depending on the type of it, there can be one or multiple pre-slots.

To connect an event post-slot to a task, the event GUID is supplied to the dependence list of the call to `ocrEdtCreate()`. An event post-slot can be connected to more than one pre-slots, and when the event is triggered, its post-slot is satisfied and the pre-slots connected to it will be satisfied.

There are several types of OCR events. Three of them are used in the HTA-OCR implementation:

1. Once event: This type of event has a single pre-slot and is triggered after the pre-slot is satisfied. The life time of a once event is the time between the call to `ocrEventCreate()` and the time it is triggered. Afterwards, it is automatically destroyed by the runtime. All subsequent attempts to satisfy it or to connect to it result in runtime errors.

2. Sticky event: A sticky event also has a single pre-slot and is triggered after its pre-slot is satisfied. But it is not automatically destroyed after one satisfaction. Instead, an explicit call to `ocrEventDestroy()` is required. Since its lifetime is explicitly controlled, it can be used whenever the objects that depend on the event do not exist when the

event is triggered.

3. Latch event: It provides a counting mechanism by using two pre-slots. The event triggers when the two pre-slots are satisfied for the same number of times. For example, if a latch event is created to count the completion of three tasks, it has to be satisfied three times at pre-slot 0 right after its creation, and then the other pre-slot will be satisfied by the three tasks for it to trigger. It is destroyed automatically by the runtime system after being triggered, same as a once event.

Data Blocks

Data blocks are where data should be stored if the data needs to have a longer lifetime and be reused by tasks other than the current one. A task can call `ocrDbCreate()` to create a data block and get back the GUID of the data block and a pointer to the starting address of the newly allocated memory space. The task can then use the data block GUID to satisfy a dependence.

When a task takes a data block in one of the pre-slots, the task can obtain the value of the starting address of the data, provided by the runtime. Since data blocks are relocatable objects, programmers should not store pointers to data blocks and pass them to other tasks in the code. The pointer values could be invalid when the task terminates.

There are different access modes for data blocks. Access mode settings give the runtime opportunities to optimize data management under the hood. For example, if a task requests for read-only access to a data block, the runtime can make a copy of it for the task, and other subsequent tasks that needs to write the same data block can start before the reader task finishes. For more details, please refer to the OCR specification [32]. In the HTA-OCR library implementation, we only use the Read-Write mode and the library makes sure that data race does not happen by enforcing ordering of task execution based on data dependences in the HTA program.

3.1.3 OCR Execution Model

The execution of OCR tasks is dictated by events. Inspired by the dataflow execution model, tasks are not scheduled for execution immediately after their creation. Instead, an OCR task stays in a queue, and the runtime system keeps track of its input dependences. When all the input dependences of an OCR task are satisfied, the task state becomes *ready* and the runtime system can schedule it.

A scheduled task is non-blocking (i.e. it runs to completion without ever being blocked), since it has all of the resources needed for the computation. As a result, forward progress is ensured. However, this poses a restriction on how tasks get input data. The only way to get input data is by, at the task creation time, specifying dependences on events which will be satisfied later with data blocks containing input data. Since a task cannot block and wait for new inputs, codes written for this execution model can have very different program structure from the popular imperative programming style.

The execution of OCR tasks depends only on data blocks passed in through its pre-slots but not on the data in the call stack or some global heap objects. This allows the runtime system to freely move tasks around, as long as the data blocks needed are also prepared on location ahead of execution. Since both tasks and data blocks are relocatable, the runtime system can make dynamic scheduling decisions for workload distribution, energy saving, and various other optimizations. This separation of concern saves application programmers from having to optimize application code with machine specific details. Also, since the OCR tasks are scheduled to run only when their preconditions are met, they never need to busy wait and consume CPU cycles. This favors having lots of tasks (much more than the number of available worker threads) in-flight so that the runtime system can keep the CPU utilization high.

For the thesis, we implemented HTA as a library on top of the OCR reference implementation built for shared memory x86 machines. When an OCR application is launched, a

configuration file is read by the runtime initialization routine and it creates a fixed number of worker threads, which are used internally by the runtime.[1] OCR worker threads use a work stealing algorithm to balance the workload and minimize idling. The reference implementation faithfully adheres the dataflow philosophy and serves as an initial platform for experimenting with so as to inspire future refinements of the OCR specification and exascale hardware architecture design.

3.2 Mapping HTA Program to OCR Tasks

An HTA program is an imperative program consisting of HTA operations and regular non-HTA operations. An implementation of the HTA programming model should execute the non-HTA operations sequentially, and parallelize the tiled array operations for maximum execution performance. In our HTA implementation on OCR (HTA-OCR), we use a single OCR task, which we refer to as the *master* task, to execute an HTA program. The master task executes the non-HTA operations sequentially till it encounters an HTA operation, where it calls the corresponding HTA library routine to dynamically determine the number of *subtasks* that should be created and the data dependences among the tasks. It then creates the tasks and specifies their dependences before it moves to the next operation.

One important observation to make is that the data dependences among tasks do not always involve the master task. In most common cases, the subtasks depend only on some tiles that are outputs of other subtasks, and they do not have data dependences due to HTA data tiles on the master task. The master task execution also does not always depend on subtask results. This means that the master task may often asynchronously create the subtasks and not wait for their completion. Based on this observation, we can imagine an HTA program as a dynamically constructed OCR task graph. It starts with a single node

[1]Note that the OCR user code does not directly interact with the threads. Instead, it deals with OCR event-driven tasks, events, and data blocks.

representing the master task, which dynamically creates new nodes that are the subtasks whenever it executes HTA operations. The subtasks can have edges pointing to each other when there are data dependences among them. As shown in Figure 3.2, the master task is represented as a thick arrow to show that its execution typically spans longer than other subtasks[2]. The blue dotted arrows represent subtask creations when the master task encounters HTA operations, and the orange arrows represent the data dependences between tasks. In this example, there are two HTA operations `OP1` and `OP2`. Some of the subtasks in `OP2` depend on the results of the subtasks in `OP1`, and our library implementation can create this task graph dynamically.

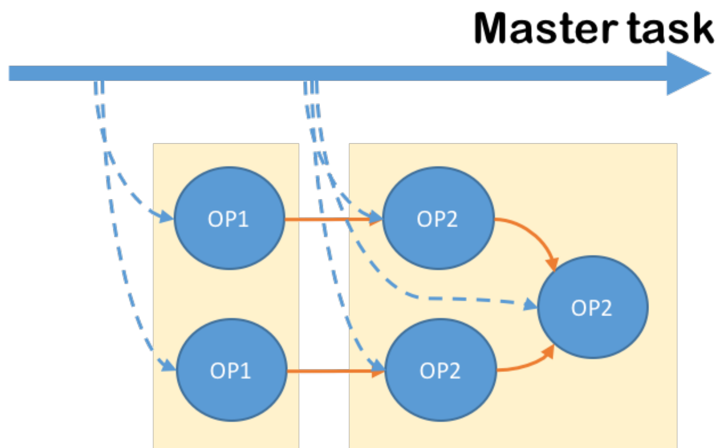


Figure 3.2: HTA-OCR program execution

Since the master task creates all other subtasks, the master task needs to figure out the dependences among them. For this reason, we add extra fields in the HTA tile metadata to record the dependence information. When the master task creates a subtask, it reads the metadata and knows which OCR events the new subtask must connect to. It then creates new OCR events to represent the completion of the new subtask, and stores their GUIDs in the tile metadata. This way, when the master task creates a subsequent subtask

[2]For the clarity of the illustrations, in the rest of this thesis, we sometimes omit showing the master task and the edges representing subtask creations in the graph.

depending on some subset of the same tiles, it can specify the correct OCR events which signify the completion of previous subtasks for the new subtask to connect to. In this way, data dependences due to HTA tiles among subtasks can be dynamically constructed in the task graph.

It is possible that the master task depends on subtask results. When it must wait for the subtasks, a split-phase continuation is performed for the master task to clone itself and setup the dependences for the clone to take subtask results. The original master task then terminates itself, and the cloned task continues when the subtask results are ready. More details about the split-phase continuation mechanism are described in Section 4.5.

Our mapping strategy converts an imperative HTA program into a dynamic task graph where the edges between nodes are data dependences. The execution of such task graphs in the OCR runtime allows the parallel subtasks to be scheduled in a more relaxed order relative to using other execution models. The opportunity for parallel subtasks to overlap with each other is automatically discovered and exploited by the runtime system, without efforts of the application programmer or a compiler static analysis. The HTA-OCR implementation details are described in Chapter 4.

The ideas presented here are simple, but the design could face some real challenges to scale up for larger systems in the real world. In Chapter 6, we further investigate the scalability issues and possible solutions for them.

Chapter 4

Implementation of HTA-OCR

We have implemented HTA-OCR, which is a C[1] library implementation of HTA programming model using the OCR programming interface for parallelization on single-node shared memory machines[2]. The OCR runtime system provides a different execution model and different mechanisms to synchronize parallel tasks than popular parallel programming models such as OpenMP and MPI. Theoretically, the new execution model can result in higher utilization of computing resources for applications with abundant asynchronous tasks [11]. Our HTA-OCR implementation dynamically coverts imperative programs into OCR task graphs in order to enjoy the benefits provided by the OCR runtime system.

4.1 Program Execution

An HTA-OCR application starts execution from `mainEdt()`, just as any other regular OCR application. Our library provides a `mainEdt()` implementation, where it sets up the envi-

[1]The reason that C language is used instead of C++ for HTA-OCR implementation is because at the beginning of the XStack Traleika Glacier project [9], in which the UIUC team worked on providing software tools for the OCR runtime system on exascale machines, there was no plan for OCR to support C++.

[2]The OCR reference implementation also supports distributed machines. However, the continuation mechanism we implement (Section 4.5) does not work in distributed environment, thus we conduct our experiments on a single-node machine. If the OCR implementation provides built-in continuation mechanism on distributed machines, our design should work properly.

ronment for an HTA-OCR application to run. Subsequently, it spawns `procEdt()` which calls `hta_main()` provided by the user and starts executing the user code. The instance of `procEdt()` is what we call the *master* task. It executes the HTA program sequentially till it meets an HTA operation, where asynchronous subtasks are created for the computation. The library routine analyzes the operands (i.e. the HTA data tiles) and determines the data dependences of the subtasks. The analysis is done by reading the dependence information, which are the OCR events stored in the HTA metadata. After the dependences are determined, the master task creates asynchronous subtasks by calling `ocrEdtCreate()` and supplying them with the OCR events for their dependence lists. The master task may immediately execute the next operation without waiting for the subtasks to complete if it does not depend on the results of the subtasks. In such case, the execution of the master task automatically overlaps with the subtask executions. Figure 4.1 shows the contrast between the library code and the user code. The calls to OCR library functions are hidden in the HTA library code.

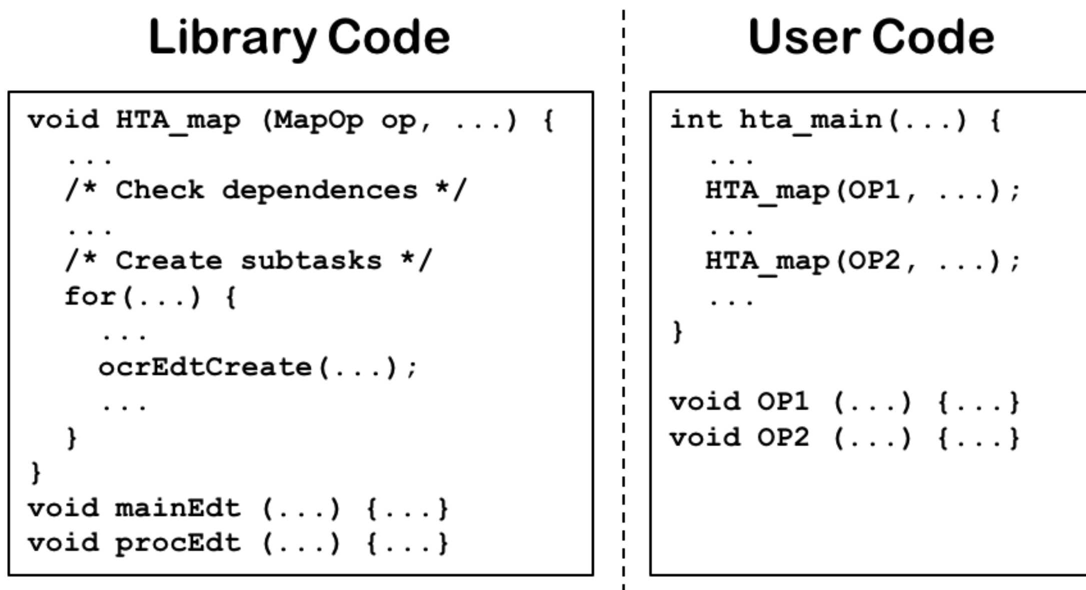


Figure 4.1: HTA-OCR library code in comparison with user program

Listing 4.1 is an HTA-OCR mergesort implementation. The code first creates an HTA `A` of height 2. Each intermediate level HTA node has two children. At Line 5, an `HTA_map()` is called on all leaf tiles in `A` to perform quick sort individually. The map operation takes an operator function pointer `QSORT` as the first argument, and a list of `ARGS` which includes `LHS` and, optionally, `RHS`, and `CAPTURE`[3]. `LHS` contains the mapped level and a list of HTAs that are on the left-hand side of the operation. For each tile at the mapped level in the `LHS`, a task is spawned to execute `QSORT` on the tile sequentially. Next, in the while loop (Line 8 - 11), the `MERGE` operation is mapped to the tiles at the intermediate levels from one level above the leaf level to the root level. Figure 4.2 shows the execution of an HTA bottom-up merge sort. The thick blue arrow represents the master task execution. It spawns asynchronous subtasks, represented in blue circles. The dotted blue arrows marked with line numbers indicate spawning of asynchronous tasks. The first four sorting tasks are spawned with input data dependences that are immediately satisfied since the tiles have not been used by other tasks yet. Next, two subtasks are spawned to merge two sorted tiles from the previous sorting tasks. The orange arrows indicate data dependences, and the data tiles flow from the sorting tasks to the merging tasks. The merging tasks start execution only when the tasks they depend on complete and the data tiles are acquired.

```
1  int  levels = 2;
2  HTA *A = /* Create HTA of tiling (2) at the root, and (2) at the next
       level. Resulting in 4 leaf tiles */
3
4  /* Quick sort the leaves individually */
5  HTA_map(QSORT, ARGS(LHS(levels, A)));
6
7  /* Bottom up merge */
8  while(levels > 0) {
9      levels--;
10     HTA_map(MERGE, ARGS(LHS(levels, A)));
11 }
```

Listing 4.1: HTA-OCR mergesort

The OCR runtime system manages multiple worker threads for tasks to be scheduled

[3]We give a detailed explanation of the map operation in Section 4.4.

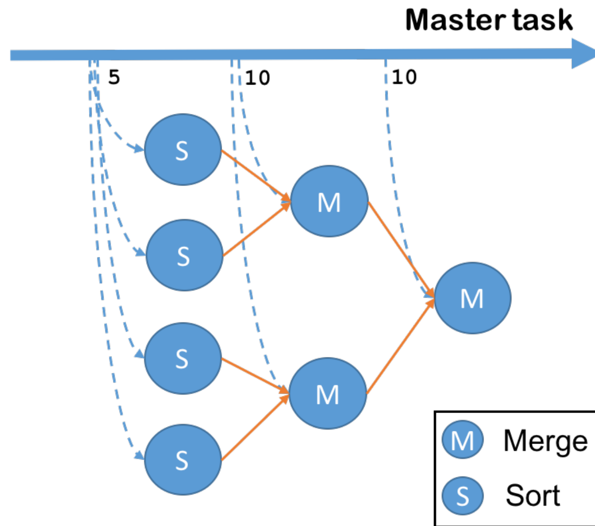


Figure 4.2: HTA-OCR mergesort program execution

to. As soon as the input dependences of a subtask are satisfied, it is ready and can be scheduled for execution. The OCR worker threads use a work-stealing scheduling algorithm so that tasks can be stolen by other worker threads to dynamically balance the workload. To best utilize computing resources, the master task should generate abundant tasks that have sparse dependence edges among them, so that there is a higher probability of having lots of ready tasks to be scheduled at any given time. When this happens, idling of worker threads would only be a result of insufficient parallelism in the program. In other words, when a program has enough parallelism exposed through using HTA operations, high utilization of computing resources is expected.

4.2 Data Dependences

In the execution style described above, parallel tasks are dynamically generated by library routines. The library also discovers the dependences among them dynamically. In general, there are only two types of tasks we need to consider: the master task, and the subtasks created for the HTA operations. Here we explain the rules of how the data dependences are

determined.

Non-HTA variables of global scope are always evaluated in the master task. On the other hand, HTAs are only accessed in the HTA operations which are executed using the subtasks spawned by the master task. Considering these types of variables used in an HTA program, data dependences exist in the following cases:

1. The computation on an HTA depends on some non-HTA variable computed previously (in terms of program order).

2. The evaluation of some non-HTA variable depends on some non-HTA variable computed previously.

3. The evaluation of some non-HTA variable depends on some HTA accessed in a previous HTA operation.

4. The computation on an HTA depends on some HTA accessed in a previous HTA operation.

In Case 1, the data dependence is resolved automatically, since the master task would have evaluated the value of the non-HTA variable at the point of spawning a task to compute the HTA. Thus, the value of the non-HTA variable can be passed by-value into the parallel subtasks, and the master task does not have to wait for it and block. An example for this would be passing a scalar coefficient value into an AXPY operation[4] performed with HTAs. Case 2 is similar in that the data dependence is guaranteed to be resolved, because non-HTA variables are always evaluated by the master task sequentially in program order.

In Case 3, since subtasks are spawned asynchronously by default, the evaluation of the non-HTA variable has data dependence on the completion of the asynchronous subtasks. The HTA library can know this type of data dependence correctly since it is the semantics

[4] $y = \alpha \times x + y$, where α is a scalar coefficient and both x and y are vectors.

of the operation. For example, if the reduction result of an HTA is assigned to a non-HTA variable, the assignment can only happen after the result is computed. Since this assignment is performed by the the master task, it needs to block and wait. OCR does not provide a built-in mechanism for a task to block and wait, so we implemented a continuation mechanism (described in Section 4.5) for this case.

In Case 4, new subtasks must be spawned by the master task to perform the operation. If the operand HTAs are not yet accessed by any previous subtasks after creation, the new subtasks can fetch the tiles as soon as they are created. If there are previous user tasks that access the tiles, they have to complete before the new tasks can start. Some mechanism is required for the new tasks to be notified when the previous user tasks are done. For this purpose we devised a tile-based dependence tracking mechanism to ensure that the tasks using the same HTAs always access them in the correct order respecting the data dependences. In short, the tile-based tracking lets the master task read the information stored in the HTA tile metadata to find out whether there is any previous user task, and the OCR event that represents the completion of previous user tasks. The master task also updates the metadata to record the new use represented by a new OCR event. The details are described in Section 4.3.

4.3 Tile-based Dependence Tracking

In the HTA-OCR execution model, all subtasks are asynchronously spawned. Subtasks may need to synchronize with each other directly without joining to the master task. To make this possible, the master task tracks the dependences of every task by using the dependence information stored in the tile metadata.

For each leaf tile, its metadata contains three OCR event GUIDs: `pre_completion_event`, `completion_event` and `latch_event`. `pre_completion_event` represents the com-

pletion of the last user task(s) accessing the tile. `completion_event` represents the completion of the new user task(s) to be spawned. `latch_event` counts the user tasks that have shared read-only access to the tile. The tile metadata also contains a `state` field. It records the tile state after being requested by the latest user task(s). There are three different states: `MODIFIED`, `SHARED`, and `UNCLAIMED`. Their meaning is described in Table 4.1.

Tile State	**Description**
`MODIFIED`	Exclusive read-write access.
`SHARED`	Shared read accesses.
`UNCLAIMED`	The tile is not claimed by any task. It could be in this state right after it is created or after an explicit `HTA_unclaim()` operation.

Table 4.1: HTA-OCR tile states.

The tile state is used by `_update_completion_event()` which is a state machine. When the master task needs to spawn a new task, it calls `_update_completion_event()` with the desired access mode of the task: read, read-write, and unclaim. Read and read-write are self-explanatory. To "unclaim" means that the task starts only when the previous user tasks are complete, but the requesting task does not need to read or write the tile. This is used to make sure computations on a tile are completed and it is useful for measuring the execution time. The `_update_completion_event()` function reads the current tile state and performs the actions defined by the state machine before the tile state transitions to the new state. Figure 4.3 shows the state machine, and the actions performed before state transitions are described using pseudo code in Table 4.2.

When the read access mode is needed, sharing of the tile should be allowed to provide more parallelism. OCR latch events are used for this purpose. An OCR latch event has two pre-slots and a single post-slot. The pre-slots can be satisfied multiple times and are associated with counters initialized to zero at creation time. Each time when a pre-slot is satisfied, the counter associated with it increments. The condition for the latch event to be triggered is when the counters of both pre-slots have equal non-zero values. In our

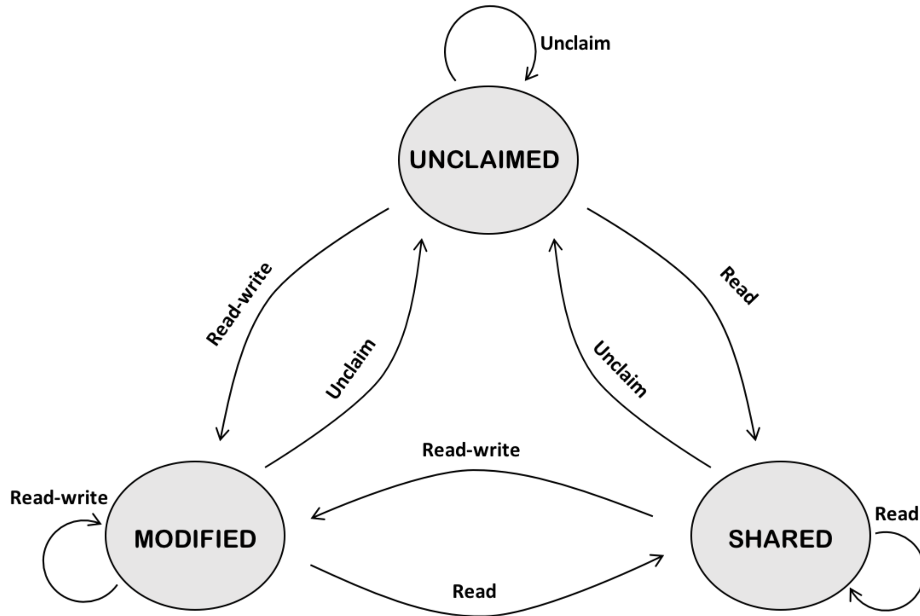


Figure 4.3: HTA-OCR tile state machine

Current State	Access mode	Next State	Actions
Unclaimed	Unclaim	Unclaimed	No action
	Read-write	Modified	`completion_event = new(event);`

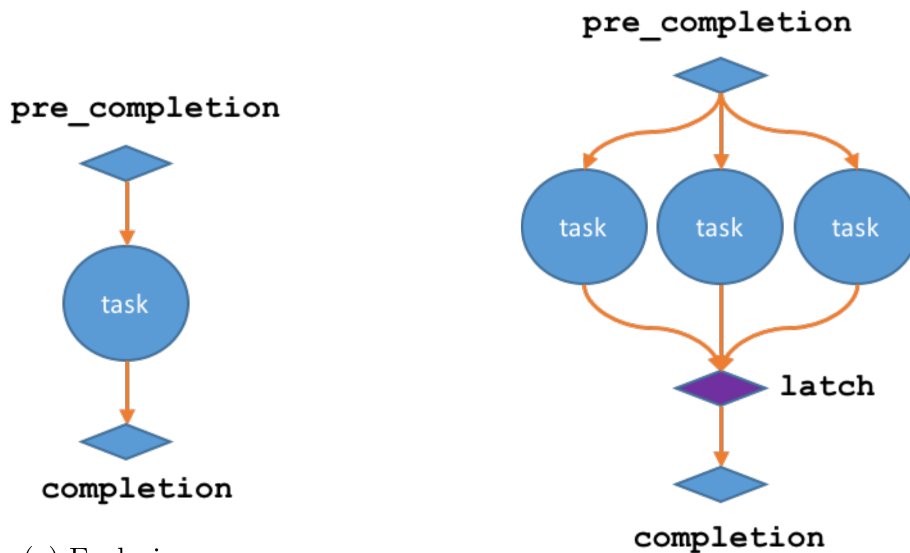
	Read	Shared	`completion_event = new(event);`
			`latch_event = new(latch event);`
			`latch_event.slot[0] += 1;`
			`latch_event → completion_event; //Link between events`

Modified	Unclaim	Unclaimed	`pre_completion_event = completion_event;` `completion_event = NULL_GUID;`
	Read-write	Modified	`pre_completion_event = completion_event;` `completion_event = new(event);`

	Read	Shared	`pre_completion_event = completion_event;`
			`completion_event = new(event);`
			`latch_event = new(latch event);`
			`latch_event.slot[0] += 1;`
			`latch_event → completion_event; //Link between events`

Shared	Unclaim	Unclaimed	`pre_completion_event = completion_event;` `completion_event = NULL_GUID;`
	Read-write	Modified	`pre_completion_event = completion_event;` `completion_event = new(event);`
	Read	Shared	`latch_event.slot[0] += 1;`

Table 4.2: HTA-OCR tile state transition table



(a) Exclusive access

(b) Shared accesses among a group of subtasks

Figure 4.4: Subgraphs generated for HTA-OCR parallel operations

implementation, when the master task discovers a first subtask requesting read access to a tile, it creates a latch event, and sets pre-slot `OCR_EVENT_LATCH_INCR_SLOT` to 1. The latch event's post-slot is connected to the pre-slot of a new sticky completion event[5]. For subsequent subtasks that also read the same tile, the master task increments the latch event's `OCR_EVENT_LATCH_INCR_SLOT`. The master task may eventually discover a subtask that needs to write the tile, and it will create a new completion event to replace the original one. All of the subtasks that have shared read accesses created in this period will increment the latch event `OCR_EVENT_LATCH_DECR_SLOT` when they complete. When all of them are done, both pre-slots have the same value, and the latch event is triggered. Figure 4.4 shows the difference between the subgraphs generated by the library for a subtask requesting exclusive access to a tile and for subtasks sharing read-only accesses to a tile. We represent the sticky

[5] A latch event is automatically destroyed after it is triggered. In our design, subsequently created subtasks could have dependence on it. Thus, a sticky event is used to retain the completion and the data. The sticky event will eventually be destroyed by the library when it determines that the completion is no longer needed for any future subtasks.

```
   /* X, Y and Z are HTAs that have
      the same dimension and tiling */
1: X = 0;
2: Y = X;
3: Z = X;
4: X += 1;
```

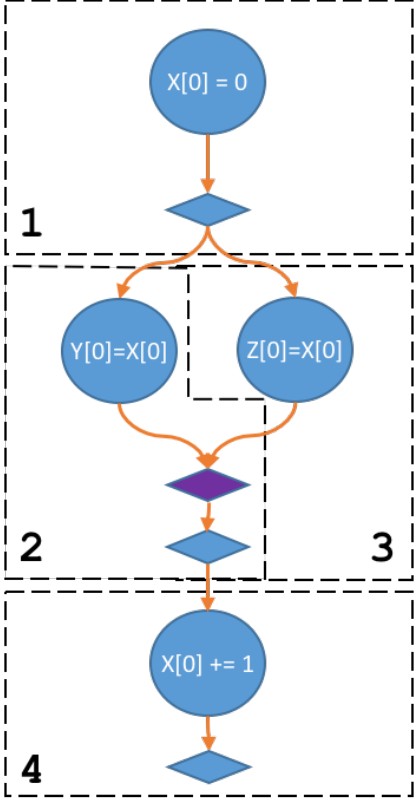


Figure 4.5: HTA-OCR task graph built with tile-based dependence tracking

events as blue diamonds and the latch event as a purple diamond.

Here, we show an example of a task graph constructed dynamically from a code segment in Figure 4.5. The graph is divided into regions with dotted borders marked by numbers representing the program statement that constructs the subgraph in the region. Three 1-D single-level HTAs `X`, `Y` and `Z` are created beforehand. Let us assume that they have the same shape and they are yet not used by any tasks after creation (i.e. they are `UNCLAIMED`). To simplify, we consider only the subgraph constructed due to the dependences of tile `X[0]`.

At Line 1, an assignment operation of constant value zero to `X` means that every scalar element in `X` will be initialized to zero. The master task spawns a task for the initialization of `X[0]`. Following the state transition table, since the access mode is *read-write* for the assignment, the `X[0]` tile state transitions from `UNCLAIMED` to `MODIFIED`, and a completion

event is created, as shown in the dotted region marked by the number 1.

Next, Line 2 is a pointwise copy operation that copies the values from X to Y. The master task creates a task to perform the copy operation `Y[0]=X[0]`. `X[0]` is on the RHS, and thus requested by the new task for *read* access mode. The data dependence due to `X[0]` is represented by a link from the previous completion event to the new task. A new latch event is created and its counter is set to 1. The state transitions from `MODIFIED` to `SHARED`.

At Line 3, another point-wise copy operation is performed and the new task `Z[0]=X[0]` requests the tile `X[0]` for *read* access again. A link is built from the previous completion event in the region 1 to the new task, and the latch event counter is incremented. No new completion event is created, since the completion of this new task is also tracked by the latch event previously created.

Finally, another operation that *reads and writes* to `X[0]` is executed at Line 4. A new task is created and it depends on the previous completion event in the region 2. A new completion event is created to for this new task `X[0]+=1`.

Notice that the OCR events and links that represent data dependences in this example are created due to a single tile `X[0]` only. If we also consider tiles `Y[0]` and `Z[0]`, there can be more tasks and dependence edges. In other words, the example only shows a subset to the whole task graph considering all tiles in all HTAs in the program.

4.4 Implementation of Core HTA Operations

In this section, the implementation of two core HTA operations are explained to help the readers better understand how the library routines determine data dependences among tasks and generate tasks. `HTA_map()` demonstrates what happens when the master task does not depend on the subtask results. In contrast, `HTA_full_reduce()` is a case when the master task depends on the subtask results and needs to perform a split-phase continuation.

4.4.1 HTA_map

The function prototype of `HTA_map()` is as the following:

```
typedef void (*HxMapOp) (HTA** lhs, HTA** rhs, uint64_t* capture);
void HTA_map(HxMapOp op, int num_args, ...);
```

In order to make `HTA_map()` operation generic enough to deal with a various number of operands, it is implemented as a C variadic function. Users need to provide a function implementation of type `HxMapOp` and pass it into `HTA_map()` as the first argument. The rest of the arguments are not directly filled in by the programmer. Instead, the programmer typically calls `HTA_map()` using a combination of macros as the following:

```
HTA_map(CALC_VELOCITY_FOR_NODES, ARGS(LHS(1, xd, yd, zd),
                                      RHS(1, xdd, ydd, zdd),
                                      CAPTURE(VCAP(dt), VCAP(u_cut))));
```

This is an example from LULESH [24] HTA-OCR implementation. Here, `HTA_map()` is called with `CALC_VELOCITY_FOR_NODES()` as the operator function. The macro `ARGS` contains a list of HTAs on the `LHS` (left-hand side) of the operator (i.e. they will be read and written), a list of HTAs on the `RHS` (i.e. they will only be read), and a list of dynamically copied 64-bit scalar values[6].

When the master task executes `HTA_map()`, it counts the number of tiles at the mapped level and checks the conformability of the operands. It determines the number of subtasks to generate, and the tiles each subtask depends on. One subtask is spawned for each tile at the mapped level of the HTA on the LHS. When there are multiple HTAs on the LHS, their shapes must be the same at the mapped level, and the counts of the tiles at the mapped

[6]The `VCAP` macro is necessary for type casting to be correct in C, the reader can ignore it and just know that the `CAPTURE` macro serves the purpose of copying dynamic scalar values and make them available for the operator function.

level for all of them is the same. In the LULESH example, if `xd`, `yd`, and `zd` are 1-D and has eight tiles each, the map operation will spawn eight subtasks.

For each of the subtasks to spawn, the master task reads the metadata of the input and the output tiles to get the GUIDs of the OCR events signifying the completion of previous subtasks that use them, and update the metadata to store event GUIDs representing the completion of the new subtask. Finally, the master task builds a dependence list and calls `ocrEdtCreate()` to asynchronously spawn the new subtask. The captured scalar values are passed by-value in the `param` list into `ocrEdtCreate()`. The master task then moves on to the other subtasks till it spawns all of the subtasks needed to perform `HTA_map()`.

When an HTA is listed on the left-hand side, the master task makes sure that the access to it in the newly generated subtask, say t, is exclusive. In other words, t can only start when previously spawned user tasks (either readers or writers) are completed, and future subtasks that access the same tile cannot start until t is done. On the other hand, if an HTA is listed on the right-hand side, it is possible that the newly generated subtask runs simultaneously with other subtasks that read the same tile. More on how tasks reading the same tiles can overlap is discussed in Section 4.3.

4.4.2 HTA_full_reduce

The `HTA_full_reduce()` operation reduces an input HTA to a scalar value. The reduction result is written back to a memory address specified by the programmer, usually it is an address on the stack. The programmer can expect the result value to be available and valid immediately after the function call. The function prototype is shown in the following:

```
typedef void (*ReduceOp)(HTA_SCALAR_TYPE stype, void* result, void* ptr);
void HTA_full_reduce(ReduceOp op, void* result, HTA *h);
```

The programmer can provide an implementation of an associative operation as the reduc-

tion operator of type `ReduceOp` or use existing ones provided by the library implementation, including `REDUCE_SUM`, `REDUCE_MIN`, `REDUCE_MAX`, and `REDUCE_PRODUCT`.

The library reads the metadata of the leaf tiles to get the OCR events the new subtasks will depend on, updates the OCR events for subsequent user tasks, and spawns parallel subtasks as many as the number of leaf tiles. Each of the subtasks performs reduction on the leaf tile assigned. The library then performs a split-phase continuation to spawn a continuation task and hook up the dependences from the parallel subtasks to it. Next, the master task terminates. The continuation task starts when all of the reduction subtasks are completed and performs sequential reduction on the received results. A reduction tree of subtasks can be used if the number of leaf tiles is very large. Since the partial reduction results are computed and received by the time the continuation task starts execution, the validity of the final reduction result is guaranteed. An example of the operation usage and more details on split-phase continuation are mentioned in Section 4.5

Since the reduction result is written to a scalar variable, the input HTA is read-only while performing the reduction. Our implementation makes it possible that the execution of the parallel reduction subtasks overlap with other tasks that read the same tiles.

4.5 Split-phase Continuation

As mentioned in Section 4.2, during program execution, it is possible that the master task needs to block and wait for the results of the subtasks. Since the master task is also an OCR task and OCR tasks are non-blocking, there is no built-in mechanism to block and take new inputs in the middle of the execution. To overcome this problem, we implemented a mechanism called split-phase continuation to allow a task to pause and resume. More specifically, we need a way for the master task to pause its execution when it needs to wait for some dynamically discovered input data dependences, and resume when the inputs are

available. Since the duration of the wait is indefinite, it is important that, during the waiting period, the master task does not occupy a worker thread (i.e. no busy waiting) so that the thread is free to do other useful work.

We let the master task terminate itself when it discovers input dependences of itself on subtasks. Before it terminates, it creates a new continuation task which is a clone of itself, and passes the original program context on. The program context includes the program stack, the register file, the program counter, and the OCR events signifying new inputs. The continuation task is just like the original master task, but with new input data dependences. Thus, the OCR runtime system will only schedule it when the input data dependences are acquired. When it starts, it restores the program context and then continues the execution with the newly received results from the subtasks. The original master task execution can be seen as a phase and the continuation as a new phase. Hence, it is called split-phase continuation, similar to the idea described in [44].

```
1 int hta_main(int argc, char** argv)
2 {
3     float result = 0.0;
4     float avg;
5     int  n = /* Initialize to total number of elements */
6     HTA* h = HTA_create( ... );
7     ... /* Initialize scalars in h */
8
9     HTA_FULL_REDUCE(REDUCE_SUM, &result, h);
10    avg = result / n;
11
12    printf("Average of all elements = %f\n", avg);
13    return 0;
14 }
```

Listing 4.2: HTA-OCR program that requires split-phase continuation

An example of a split-phase continuation is shown in Listing 4.2. `hta_main()` first creates an HTA `h` at Line 6 and then initializes it. Then, at Line 9, the library routine `HTA_FULL_REDUCE()` is called on `h`, with the reduce operator `REDUCE_SUM` for summation, and an address on the stack `&result` to hold the reduction result. Since the reduction operation writes the output to a stack variable, this is an example of Case 3 described in Section 4.2

when a split-phase continuation is required.

HTA_FULL_REDUCE() is a macro that contains a call to the library routine HTA_full_reduce() and some extra code for processing the split-phase continuation. The split-phase continuation happens at Line 9, and the original master task terminates. When the continuation task receives the new reduction results from subtasks, the execution continues at Line 9. Assuming there are few tiles, a sequential reduction of the partial results is performed before moving on to Line 10, where a division is performed to get the average value. The execution then proceeds to Line 12 to print out the average value.

The task graph formed by the program execution can be found in Figure 4.6. Suppose there are three leaf tiles in h, the master task spawns three subtasks and each of them sequentially performs reduction on its assigned leaf tile to get a scalar value. Since the number of subtasks is small, the three scalar values are sent directly to the continuation task to be reduced sequentially (shown as the small blue circle) to the final result. A tree reduction can be used if the amount of leaf tiles are large, so more parallelism is exploited.

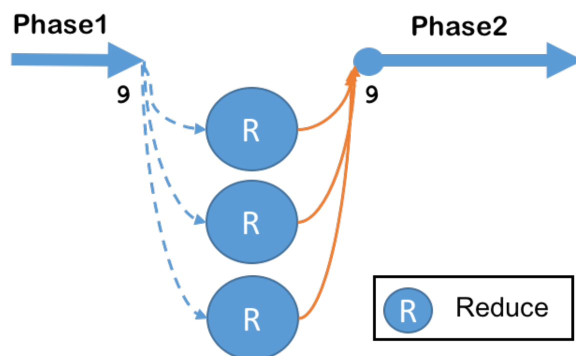


Figure 4.6: Split-phase continuation

Split-phase continuations can have negative impact on the performance. Other than the overhead of cloning the program context and creating a new task, the major issue is that the master task execution is paused. As mentioned previously, it is crucial for applications running on top of the dataflow runtime system to generate lots of tasks so that the runtime

system can keep worker threads busy. A split-phase continuation pauses the master task who generates parallel subtasks. This could lead to temporary depletion of work and causes idling. Programmers should be fully aware of the consequences of invoking the operations that require continuations and only use them when it is absolutely necessary for the program correctness.

A way to avoid having split-phase continuations is to remove the data dependence on subtasks from the master task. For example, if the reduction result is not needed directly by the master task, it is possible to implement a reduction operation that takes a single-element HTA as the output storage. This operation can store the reduction result to the single-element HTA in a subtask, without the involvement of the master task. And if there are subsequent operations depending on the reduction result, they would request the single-element HTA as their input data dependences. Even if the master has to get the reduction result, the read-back from the single-element HTA can be postponed as late as possible to allow more subtasks of other operations to be generated, so that the system is not depleted of work.

4.6 Prioritized Tasks

At any given moment during the program execution, there can be more ready tasks than available worker threads. The runtime system uses heuristics to schedule ready tasks. For example, the default OCR scheduler use LIFO policy to schedule the newest local task in the ready queue first, and FIFO policy while stealing tasks from remote. Simple heuristic scheduling algorithms can often fail to produce optimal schedule and resulting in inefficiency, since they do not consider the task granularity or the parallelism the can be enabled by executing critical tasks.

On the other hand, the programmer might have a better knowledge to estimate the task

Constant Name	Value
HTAOCR_EDT_TOP_PRIORITY	ULONG_MAX
HTAOCR_EDT_MEDIUM_PRIORITY	ULONG_MAX >> 1
HTAOCR_EDT_DEFAULT_PRIORITY	HTAOCR_EDT_MEDIUM_PRIORITY
HTAOCR_EDT_LOWEST_PRIORITY	0

Table 4.3: HTA-OCR task priority preset values.

granularity and provide guidance for the scheduler. In OCR, other than the default scheduler, a priority scheduler is provided. It takes the user assigned task priority into consideration while making scheduling decisions. When tasks are created with priority hints, the priority scheduler always schedules the ready tasks with the highest priority first. We create wrapper functions in HTA to expose the priority assignment capability to users. In our experiments, assigning priorities in the application level can improve performance over using heuristics.

The API for setting task priorities is `HTAOCR_SET_PRIORITY(p)`. It sets the priority to an unsigned integer value `p` for all tasks generated subsequently. The higher the value of `p` is, the higher the priority. The library also provide pre-defined values for programmers when they only need a few different priorities as shown in Table 4.3.

4.7 Tracing API

For debugging and performance analysis purposes, we include a set of functions that can generate a trace for HTA-OCR program execution. To utilize them, the programmer has to include headers `HTA_trace.h` and `HTA_timer.h`, put `HTA_trace_init()` in the setup code, and put `HTA_trace_finalize()` at the point where the trace should be dumped, usually before program terminates. To trace individual tasks, the programmer defines an integer value as the type of the task, and inserts calls to `HTA_insert_trace_entry()` at the point where the task starts and right before it ends. See Listing 4.3 for an example. Line 3 and 14 are calls to insert entries for the start and the end of `INIT_STRESS_TERMS`. The overhead of

the calls should be easily amortized by the actual work of the task, since the calls are quick memory stores to the internal memory space managed by the tracing library.

The dumpped `trace.log` file contains the start time and end time of the tasks. We use the `trace_viewer` program from the SWARM project [30] to display it in a comprehensive manner, as shown in Figure 4.7. Each colored bar represents the execution of a task, and tasks of the same type are shown in the same color. Each row represents the trace of a worker thread, and the white space in each row means either the tasks being executed in that duration are not recorded, or the worker thread idles.

```
1 void INIT_STRESS_TERMS(HTA** lhs, HTA** rhs, uint64_t* capture)
2 {
3     HTA_insert_trace_entry(HTA_get_thread_num(), 1, INIT_STRESS_TASK,
         gettime_ns());
4     Index_t numElem = lhs[0]->flat_size.values[0];
5     Real_t *sigxx = HTA_get_ptr_raw_data(lhs[0]);
6     Real_t *sigyy = HTA_get_ptr_raw_data(lhs[1]);
7     Real_t *sigzz = HTA_get_ptr_raw_data(lhs[2]);
8     Real_t *p = HTA_get_ptr_raw_data(rhs[0]);
9     Real_t *q = HTA_get_ptr_raw_data(rhs[1]);
10
11    for (Index_t i = 0 ; i < numElem ; ++i){
12       sigxx[i] = sigyy[i] = sigzz[i] =  - p[i] - q[i] ;
13    }
14    HTA_insert_trace_entry(HTA_get_thread_num(), 0, INIT_STRESS_TASK,
         gettime_ns());
15 }
```

Listing 4.3: An operator function with tracing calls

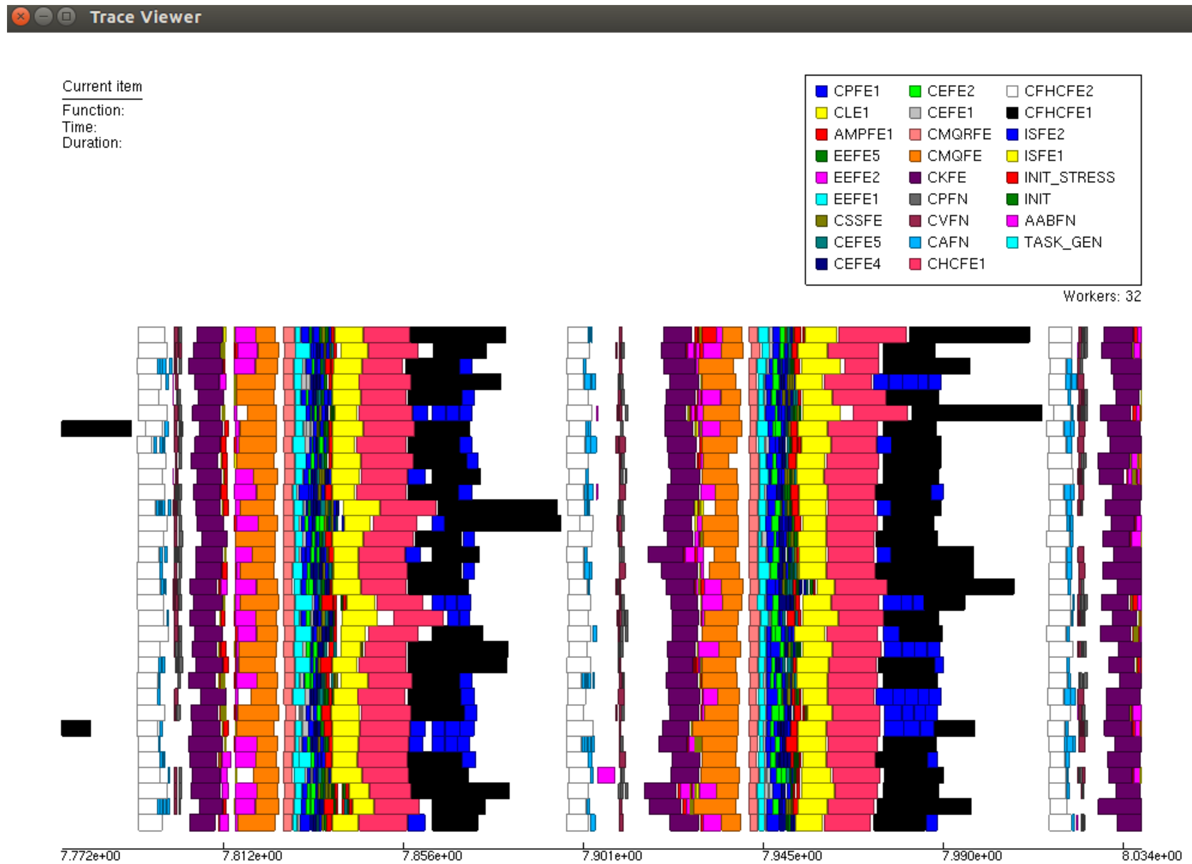


Figure 4.7: Program trace displayed by SWARM trace_viewer application

Chapter 5

Performance Evaluation

In this chapter, we present the performance evaluation of the HTA C library implementation on top of OCR (HTA-OCR) using several benchmark programs, including hand-coded parallel algorithms and some benchmarks from Coral [29], Rodinia [13] and NAS Parallel Benchmarks [4]. We describe the problems that the benchmark programs solve and the parallel algorithms used. The performance results of the HTA-OCR versions are compared with their OpenMP counterpart [1], because OpenMP is widely used for parallelizing applications on shared memory machines and the benchmarks used all have OpenMP implementations. The programmability of OpenMP is also similar to that of HTA. Based on the experimental results, we analyze the reasons for the differences in their performances.

The experiments are conducted on a single node equipped with four Intel Xeon E7-4860 processors, each of which has ten cores. We tested the scalability of our implementation using up to forty OCR worker threads in hope that each thread is assigned to a dedicated core so that hyperthreading effects can be avoided. Our purpose is to compare the execution models, so we timed the execution of the major computation in the benchmarks, excluding initial setup such as initial array allocations and input data generation.

[1]For a few benchmarks, we also include comparisons with pure OCR implementations

5.1 2-D Convolution

Convolution is an important method for signal and image processing. 2-D convolution takes an input matrix which may represent an image and the other matrix (often referred to as filter) performs stencil computation and produces an output matrix. Each element in the output matrix can be computed by performing stencil computation which reads all elements in the filter and the same amount of elements in the input matrix. As shown in Figure 5.1, when the filter is a 3×3 matrix, the 9 points in the filter and 9 points in the input matrix are used in the computation of one point in the output matrix. The stencil computation of each point of the output matrix is entirely independent and thus the algorithm can be parallelized easily by decomposing the output matrix.

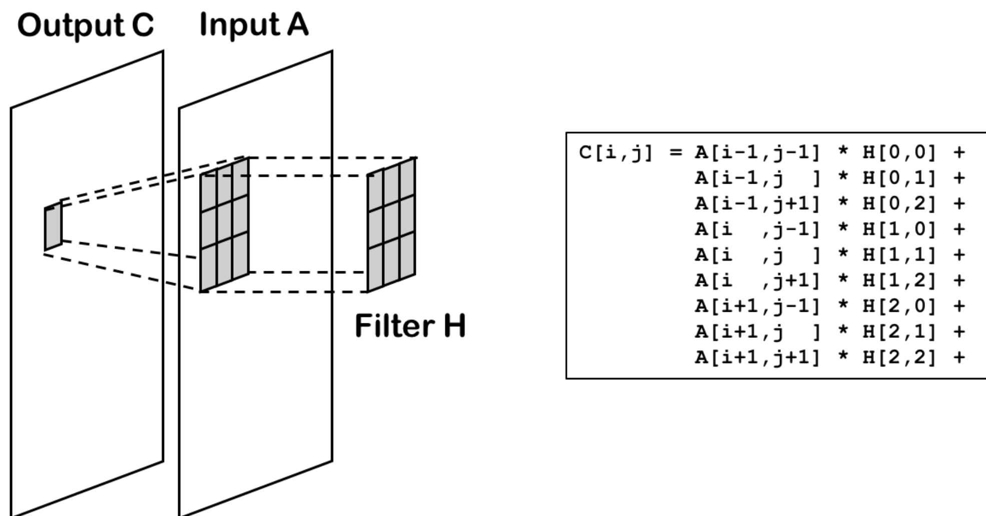


Figure 5.1: Single output element computation of convolution 2D with filter size 3×3

The HTA-OCR implementation of the major computation in 2-D convolution (Listing 5.1) is performed by call to `HTA_map()` and the operator function `TILE_CONV2D` takes a tile from the input matrix `A` and the filter `H` which is one-level and single-tiled in the RHS, and a tile from the output matrix `C` in the LHS. Executing the program, the map function generates as many subtasks as the number of tiles in `C` and all of them can run in parallel.

In contrast, in the OpenMP implementation, a two-level nested for loop is collapsed into a single parallel for loop that iterates over all tiles in the output matrix. An output tile is computed by one iteration of the parallel for loop.

```
1 void  conv2d(HTA* A, /*  Input  matrix  */
2              HTA* H, /*  Filter  */
3              HTA* C  /*  Output  matrix  */ ) {
4     HTA_map(TILE_CONV2D , ARGS(LHS(1, C), RHS(1, A), RHS(0, H)));
5 }
```

Listing 5.1: Convolution 2D in HTA-OCR

```
1 /* A: Input matrix, H: Filter, C: Output matrix */
2 /* Tile control loop collapsed * /
3 #pragma omp parallel for schedule(runtime)
4 for(int t = 0; t < num_tiles; t++) {
5     int ti = t / num_col_tiles;
6     int tj = t % num_col_tiles;
7
8     TILE_CONV2D(C, A, H, ti, tj);
9 }
```

Listing 5.2: Convolution 2D in OpenMP

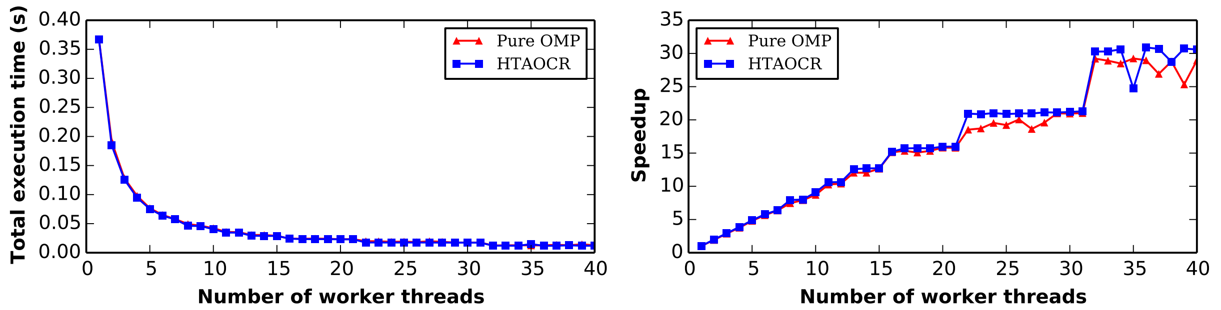
The data layout of the input matrix is different in the two versions. In HTA-OCR, the input matrix is tiled and each tile has extra padding to store elements that are needed for the stencil computation to avoid memory accesses to neighboring tiles. In OpenMP, the input matrix is seen as globally available and can be randomly accessed.

The two versions show comparable performances for different configurations as shown in Figure 5.2. The input matrix is 2048×2048 partitioned into either 8×8 tiles or 16×16 tiles to see whether the tiling affects the scalability. Two different filter sizes, 7×7 and 11×11, are used with each tiling configuration, but the difference does not seem to have a significant impact on the results. For 8×8 tiles, a noticeable staircase pattern can be observed in the speedup curves of both versions. It also shows when the tiling is 16×16 but is less obvious. This is because the tasks are not evenly divided among threads, even though the workload of each task is the same. Suppose it takes time t for a task to complete, the total execution

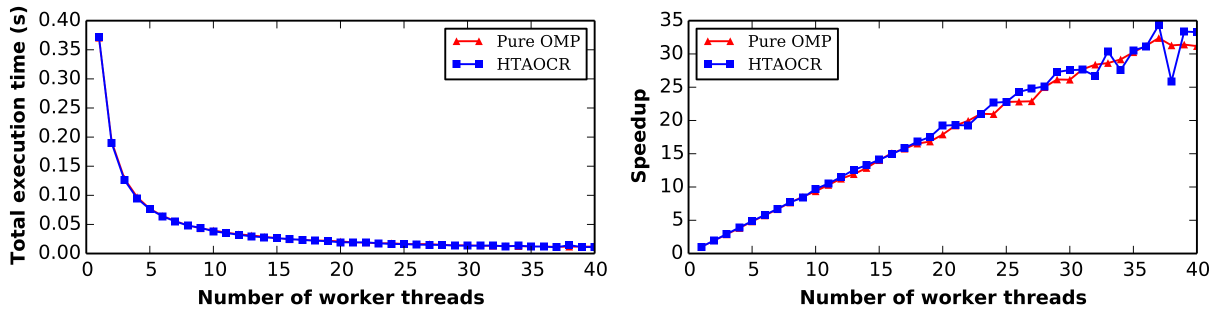
time is:

$$T = t \times \lceil \frac{Number\ of\ tasks}{Number\ of\ threads} \rceil$$

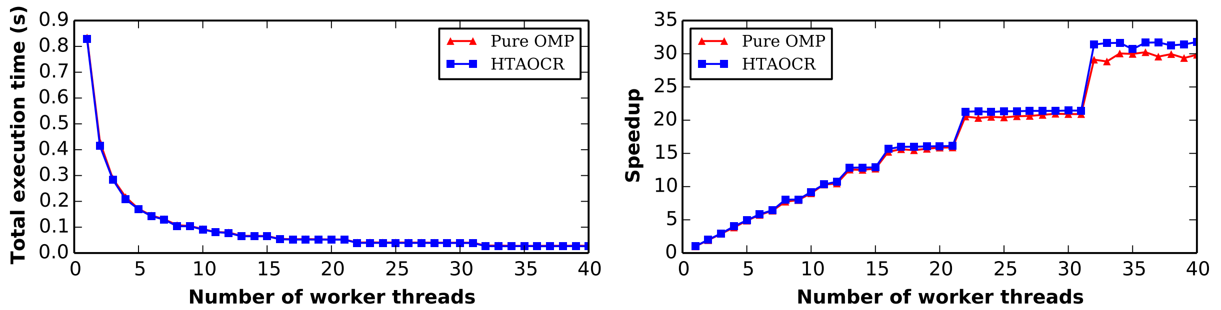
For a certain range of thread counts, the formula evaluates to the same value. Increasing the number of tiles alleviates this problem, but we also have to be careful not to use too many tiles for the reason that the overhead of task generation may be too high.



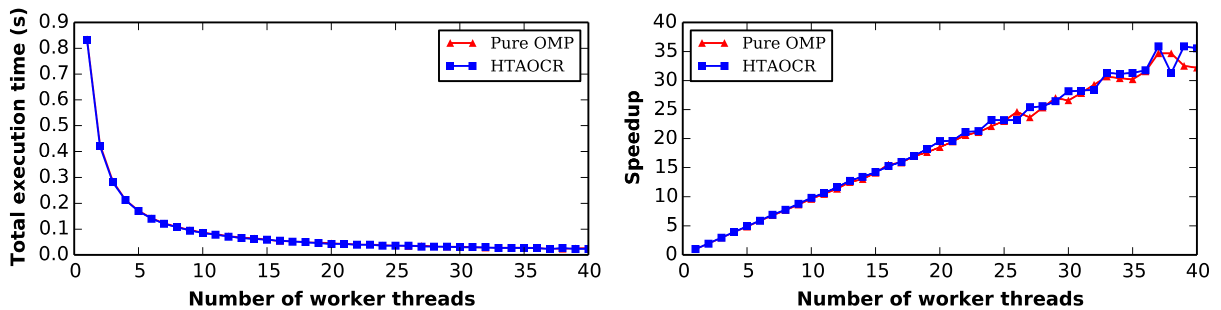
(a) 8×8 leaf tiles, 7×7 filter



(b) 16×16 leaf tiles, 7×7 filter



(c) 8×8 leaf tiles, 11×11 filter



(d) 16×16 leaf tiles, 11×11 filter

Figure 5.2: 2-D Convolution results, 2048×2048 elements

5.2 Tiled Matrix-Matrix Multiplication

Listing 5.3 is the HTA implementation of the tiled matrix-matrix multiplication algorithm using a three-level nested loop. In each iteration of the innermost loop body, tuples are created (Line 6-8) for selecting tiles in the tiled matrices. Each call to `HTA_map()` (Line 9-12) creates a single asynchronous subtask which executes `MATMUL`, the kernel function that performs sequential matrix-matrix multiplication on tiles. The library requests read-write access for the operand `C[i][j]` listed in `LHS` and read accesses for `A[i][k]` and `B[k][j]` listed in `RHS`.

```
1  /* A, B: Input tiled matrix, C: Output tiled matrix */
2  /* TILES: Number of tiles along a single dimension  */
3  for(k = 0; k < TILES; k++) {
4    for(i = 0; i < TILES; i++) {
5      for(j = 0; j < TILES; j++) {
6        Tuple ij = Tuple_create(2, i, j);
7        Tuple ik = Tuple_create(2, i, k);
8        Tuple kj = Tuple_create(2, k, j);
9        HTA_map(MATMUL,
10               ARGS(LHS(0, HTA_pick_one_tile(C, &ij)),   /* C[i][j] */
11                    RHS(0, HTA_pick_one_tile(A, &ik),    /* A[i][k] */
12                           HTA_pick_one_tile(B, &kj))); /* B[k][j] */
13     }
14   }
15 }
```

Listing 5.3: Tiled matrix-matrix multiplication in HTA-OCR

We assume the matrices `A`, `B` and `C` are square and tiled the same way. In each `k` loop

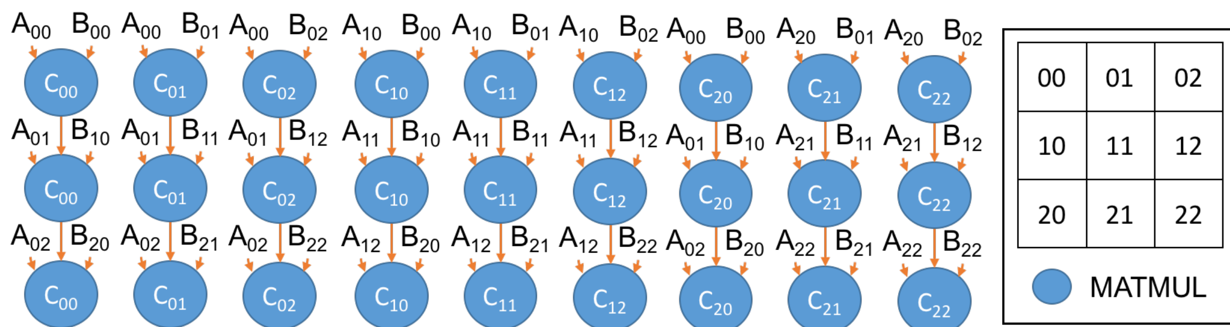


Figure 5.3: Task graph of tiled matrix-matrix multiplication of 3×3 tiles

iteration, the new multiplication result of `A[i][k]` and `B[k][j]` is accumulated to `C[i][j]`. All the tiles in `A` and `B` are read-only after initialization. Hence, all the parallel subtasks only request read accesses and can access them simultaneously. In contrast, the tasks accessing the same `C` tile form a dependence chain due to the flow and output dependences. See Figure 5.3 for an example dataflow task graph when input matrices are 3×3 tiles.

Listing 5.4 is the OpenMP implementation used in the experiments. `A`, `B` and `C` are arrays of `struct TILE`, which contains a pointer to the raw data. This version has the same nested loop structure as the HTA-OCR version, except that the innermost two loops are collapsed into a single `ii` loop so that the parallelism is maximized. Notice that there is an implicit barrier at the end of `ii` loop at Line 9. As a result, the program executes in a bulk-synchronous fashion.

```
1  /* A, B: Array of input tiles, C: Array of output tiles */
2  /* TILES: Number of tiles along a single dimension      */
3  for(int k = 0; k < TILES; k++) {
4    #pragma omp parallel for
5    for(int ii = 0; ii < TILES*TILES; ii++) {
6      int i = ii / TILES;
7      int j = ii % TILES;
8      MATMUL(&C[i*TILES+j], &A[i*TILES+k], &B[k*TILES+j]);
9    }
10 }
```

Listing 5.4: Tiled matrix-matrix multiplication in OpenMP

For this benchmark, we also compare to a hand-coded pure OCR implementation. The complete pure OCR implementation is included in Appendix A due to its lengthiness. The OCR version first sequentially builds the whole task graph as shown in Figure 5.3 and then starts the first round of tasks by explicitly satisfying the input dependences of them. The rest of the tasks start when their dependences are satisfied with the required tiles. Compared with the HTA-OCR version, the pure OCR version has about $2.3\times$ more lines.

The results of the three implementations using a naive hand-coded sequential matrix multiplication kernel are shown in Figure 5.4. We use matrices of size 1600×1600 and

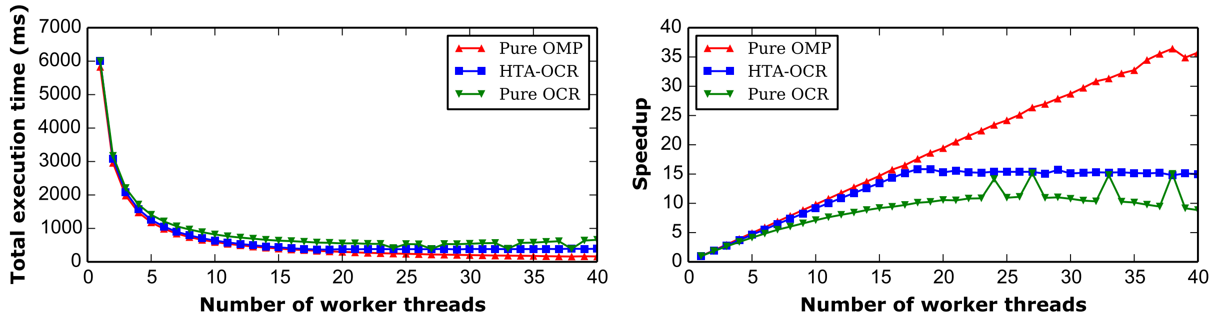
3200×3200 with varying tile sizes 50×50, 100×100 and 200×200 to observe the effects of different combinations of the problem size and the tiling. In 5.5a, the HTA-OCR version stopped scaling when thread count is larger than 18. This is because the task spawning overhead dominates the execution time, and adding more threads does not help. In contrast, the pure OCR version shows inferior performance, because it spawns the tasks sequentially to build the complete task graph before computation starts, while the HTA version can overlap task spawning and computation.

In the experiment corresponding to Figure 5.5b, where we keep the problem size fixed while reducing the number of tiles, both the pure OCR and the HTA-OCR results improve because there are fewer tasks to generate and the larger task granularity amortizes the overhead better. The OpenMP version performs slightly worse than the first configuration, because the workload can not be perfectly balanced between the implicit barriers, resulting in some idle time of the threads. In contrast, both the OCR and HTA-OCR versions can schedule tasks dynamically and there are enough tasks to keep threads busy most of the time. Also, if the task spawning time of the pure OCR version is excluded, it will have comparable performance as the HTA version. In 5.5c and 5.5d, a larger problem size is used. When tiles are 100×100, the pure OpenMP and the HTA-OCR versions have similar results, while the pure OCR version still falls behind due to its sequential task spawning. When tiles are 200×200, OpenMP speedup curve shows the same staircase pattern due to the task distribution problem as described in Section 5.1, while the other two versions have smoother speedup curves.

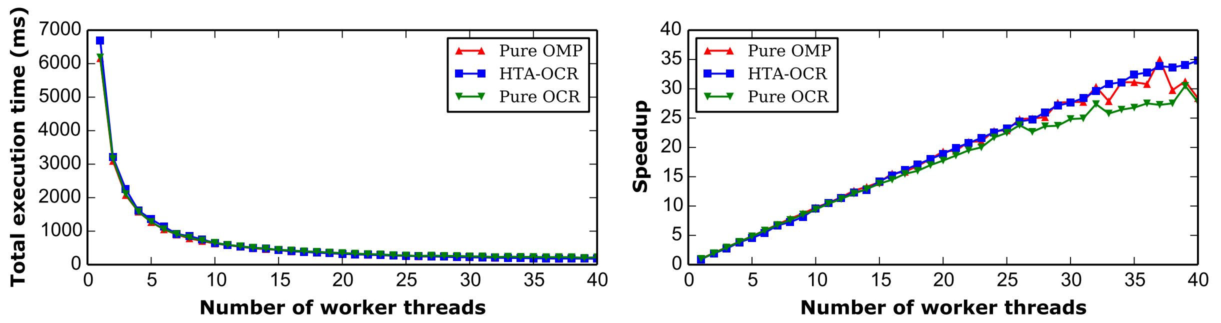
We also conduct the experiments using sequential MKL [25] `dgemm` kernel in place of the naive kernel. Besides the three versions, we also measured the execution time of a version computing the untiled matrix-matrix multiplication using multithreaded MKL `dgemm` kernel, as shown in Figure 5.5. Since the sequential MKL `dgemm` kernel is faster than the naive kernel, with the same tiling and problem size as the previous experiments, it can be seen

as a reduction of task granularity. The speedup curves are plotted using the execution time using single thread MKL `dgemm` execution time as base.

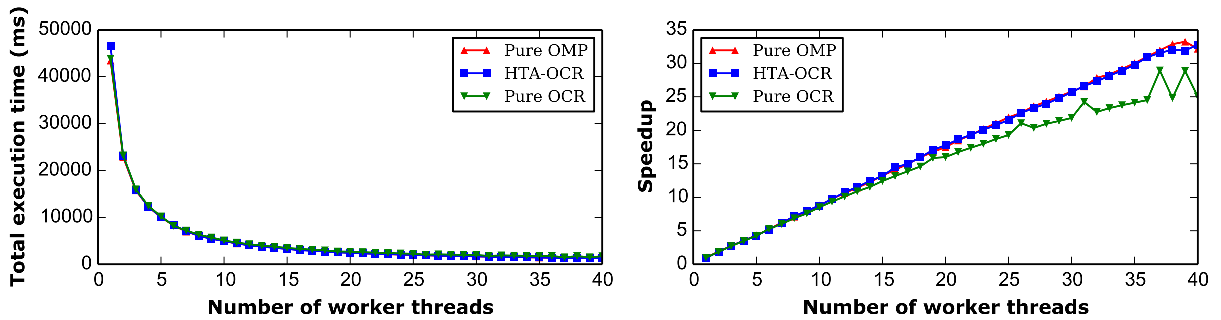
Comparing the pure OpenMP, the pure OCR and the HTA-OCR versions, they are comparable only when the tile size is large (200×200). When the tile size is smaller, the OpenMP version performs the best while the pure OCR and the HTA-OCR versions suffer from task overhead. Pure MKL multithreaded results are plotted as a comparison. They did not outperform the tiled versions and some more fine-tuning of MKL parameters might be needed to get better results.



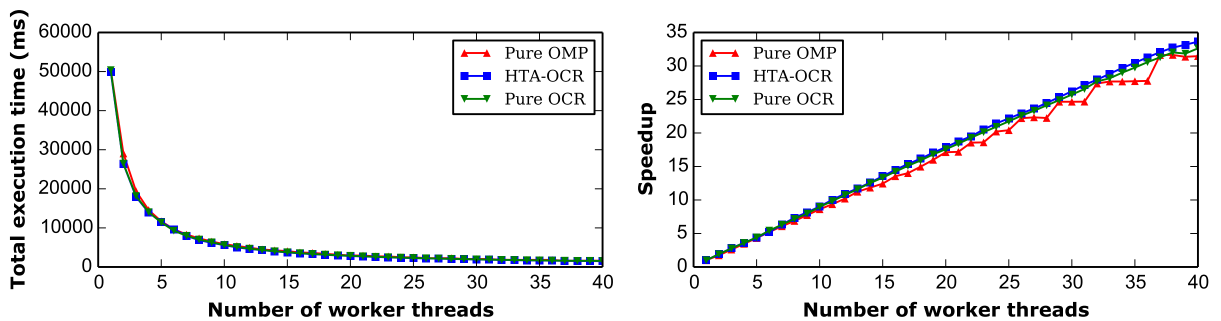
(a) 32×32 tiles, 50×50 elements per tile



(b) 16×16 tiles, 100×100 elements per tile

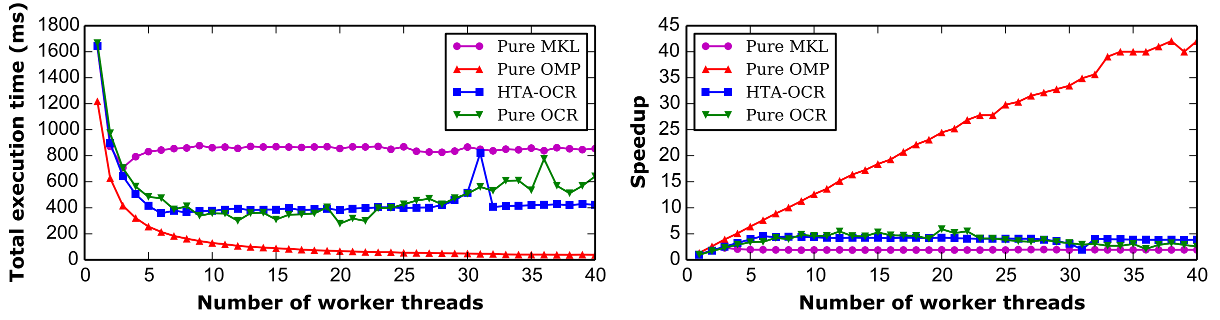


(c) 32×32 tiles, 100×100 elements per tile

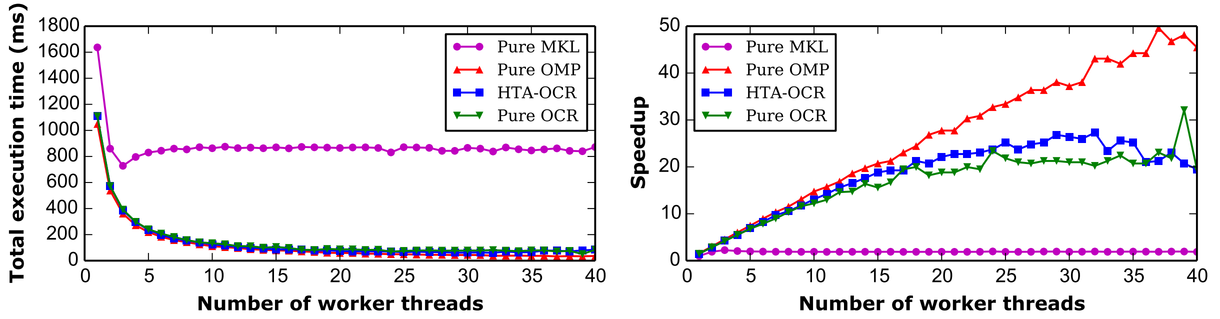


(d) 16×16 tiles, 200×200 elements per tile

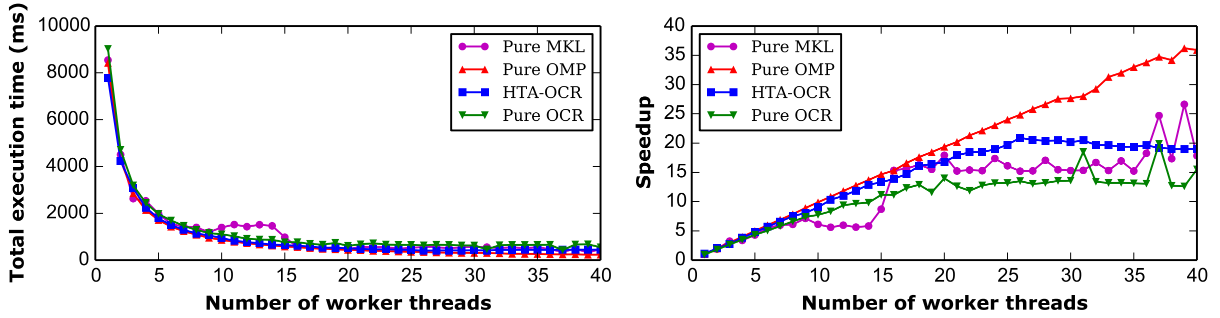
Figure 5.4: Tiled matrix multiplication results using naive kernel



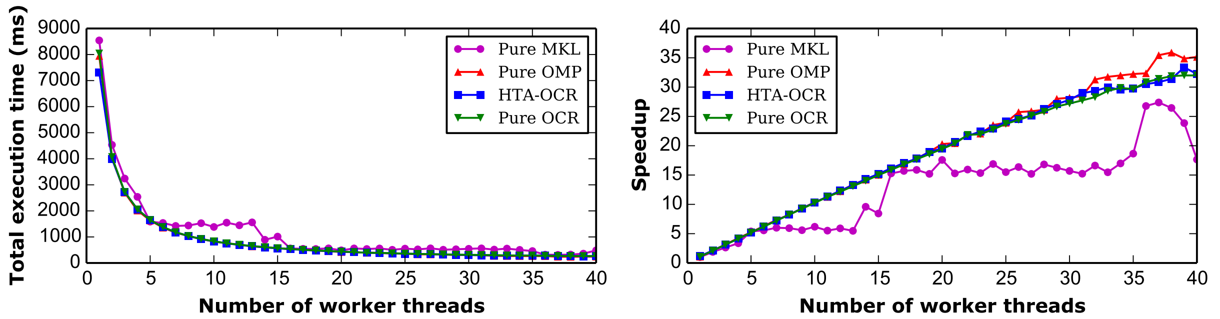
(a) 32 × 32 tiles, 50 × 50 elements per tile



(b) 16 × 16 tiles, 100 × 100 elements per tile



(c) 32 × 32 tiles, 100 × 100 elements per tile



(d) 16 × 16 tiles, 200 × 200 elements per tile

Figure 5.5: Tiled matrix multiplication results using MKL `dgemm` kernel

5.3 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. It is frequently used in solving numeric systems. Since the input matrix is symmetric, the computation can be applied to one triangular matrix and transpose the result to get the other half.

```
1  /* hA, hB:  Input tiled matrix, hC: Output tiled matrix */
2  /* nBlocks: Number of tiles along a single dimension    */
3  for(int k = 0; k < nBlocks; k++) {
4    Tuple kk = Tuple_create(2, k, k);
5    HTA *hAkk = HTA_pick_one_tile(hA, &kk);
6    HTA_map(POTRF, ARGS(LHS(0, hAkk)));
7    for(int i = k+1; i < nBlocks; i++) {
8      Tuple ik = Tuple_create(2, i, k);
9      HTA *hAik = HTA_pick_one_tile(hA, &ik);
10     HTA_map(TRSM, ARGS(LHS(0, hAik), RHS(0,hAkk)));
11   }
12   for(int j = k+1; j < nBlocks; j++) {
13     Tuple jj = Tuple_create(2, j, j);
14     Tuple jk = Tuple_create(2, j, k);
15     HTA *hAjk = HTA_pick_one_tile(hA, &jk);
16     HTA *hAjj = HTA_pick_one_tile(hA, &jj);
17     HTA_map(SYRK, ARGS(LHS(0, hAjj), RHS(0, hAjk)));
18     for(int i = j+1; i < nBlocks; i++) {
19       Tuple ij = Tuple_create(2, i, j);
20       Tuple ik = Tuple_create(2, i, k);
21       HTA *hAij = HTA_pick_one_tile(hA, &ij);
22       HTA *hAik = HTA_pick_one_tile(hA, &ik);
23       HTA_map(GEMM, ARGS(LHS(0, hAij), RHS(0, hAik, hAjk)));
24     }
25   }
26 }
```

Listing 5.5: Tiled dense Cholesky factorization in HTA-OCR

Here we use the tiled version of the Cholesky fan-out algorithm with the MKL [25] kernels shown in Listing 5.5. In each `k` loop iteration, the algorithm first performs Cholesky factorization (calling `POTRF`) on the diagonal tile `A[k][k]` (line 4 - 6). Next, using `A[k][k]` as its input, the first inner `i` loop applies `TRSM` to the `k`th column tiles below the diagonal tile (line 7 - 11). Since `TRSM` tasks of the same `k` iteration request read access to `A[k][k]` and write accesses to different tiles in the `k`th column, they can all execute in parallel to

each other. The submatrix update is performed in the inner `j` loop by calling `SYRK` (line 13 - 17) or `GEMM` (line 18 - 23) depending on whether the output tile is on the diagonal. The submatrix updates can all be done in parallel. The inner loop iteration count decreases when `k` goes up. As a result, the parallelism is the largest at the beginning of the execution and diminishes towards the end.

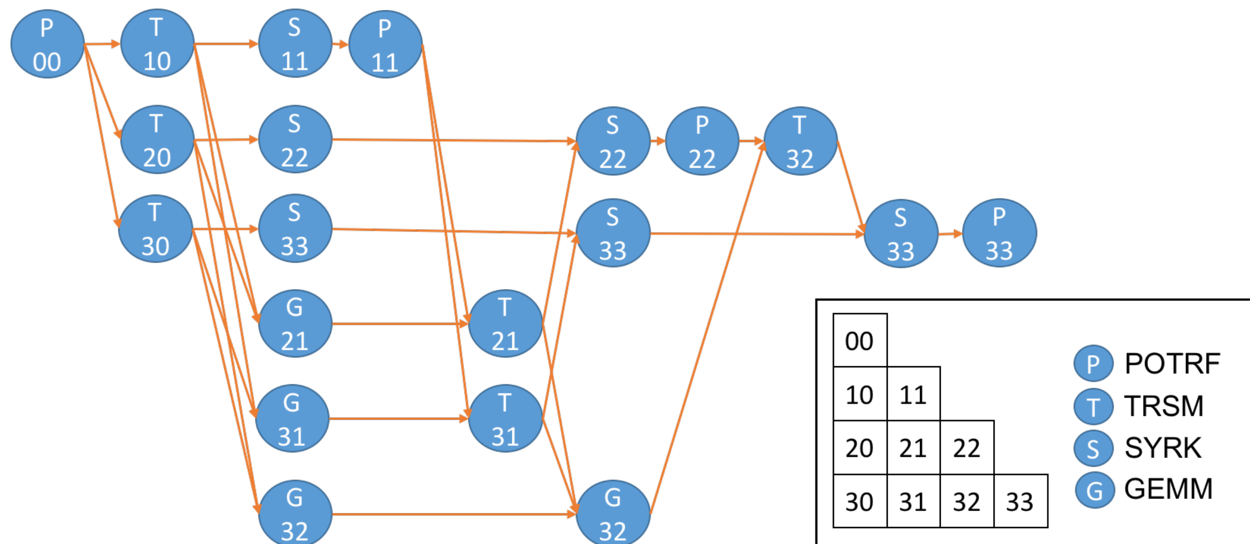


Figure 5.6: Task graph of tiled Cholesky factorization of 4×4 tiles

The data dependences form a complex dataflow task graph, as shown in Figure 5.6. The tasks are marked with the type of the operation and the indices of their output tile. The parallelism grows rapidly initially and diminishes with the progress of the execution. The task dependences become even more complex when the amount of tiles increases. When data dependences are complicated, programmers typically formulate algorithms using bulk-synchronous methods. However, using global barriers is often an overkill. Whether a task can start should only depend on the readiness of the data tiles needed for its computation. As pointed out in [12], dynamic scheduling based on task dependences can relax the order of task executions and eliminate the idle time that happens in the bulk-synchronous execution of the Cholesky factorization algorithm. Our implementations in both pure OCR and

HTA-OCR generate exactly the graph in Figure 5.6 by using OCR events to represent data dependences among tasks with no global synchronization barriers required. Observing the task dependences, we expect not only the tasks of the same iteration to overlap, but the tasks across iterations should also overlap. Both the complete OCR implementation and the HTA-OCR one can be found in Appendix B. Compared with the HTA-OCR version, the pure OCR program has about 1.9× more lines.

```
1  /* A, B: Array of input tiles, C: Array of output tiles */
2  /* nBlocks: Number of tiles along a single dimension      */
3  for(int k = 0; k < nBlocks; k++) {
4    int numGEMMS = (nBlocks-k)*(nBlocks-k-1)/2;
5    POTRF(&A[k*nBlocks+k]);
6    #pragma omp parallel for schedule(runtime)
7    for(int i = k+1; i < nBlocks; i++) {
8        TRSM(&A[i*nBlocks+k], &A[k*nBlocks+k]);
9    }
10   #pragma omp parallel for schedule(runtime)
11   for(int x = 0; x < numGEMMS; x++)
12   {
13     int i, j;
14     GET_I_J(x, k+1, nBlocks, &i, &j);
15     if(i == j)
16       SYRK(&A[j*nBlocks+j], &A[j*nBlocks+k]);
17     else
18       GEMM(&A[i*nBlocks+j], &A[i*nBlocks+k], &A[j*nBlocks+k]);
19   }
20 }
```

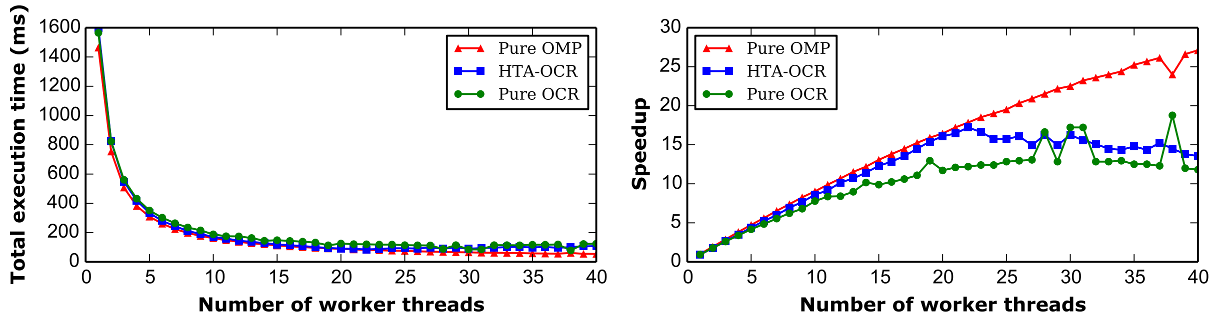
Listing 5.6: Tiled dense Cholesky factorization in OpenMP

As a comparison, the OpenMP implementation is shown in Listing 5.6. Essentially, it has the same program structure as the HTA-OCR version, except for the differences in the parallel constructs. Dynamic scheduling is used to alleviate load imbalance in the experiments. Although the HTA-OCR version has more lines of code, conceptually it is not more complicated than the OpenMP version. If the library is implemented in C++, with C++ operator overloading, the HTA program can definitely be more concise.

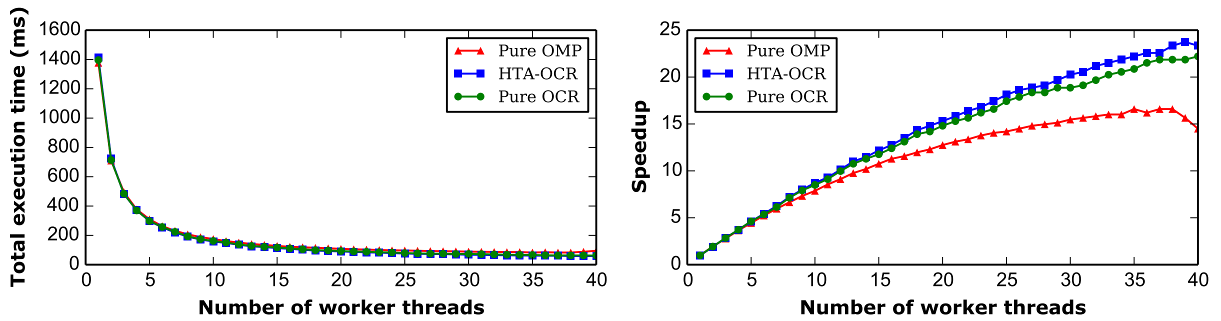
Figure 5.7 contains experimental results of pure OCR, pure OpenMP and HTA-OCR with four different configurations. In Figure 5.7a when there are many tiles and the tile size

is small, pure OpenMP performs the best and pure OCR performs the worst. The HTA-OCR implementation stops scaling when thread count is larger than 22. This is the same problem due to task generation overhead mentioned in Section 5.2, and the pure OCR version is worse than the HTA-OCR one also because of the sequential task graph generation. In Figure 5.7b, the same problem size is used but the tiling is changed to have fewer tiles and larger task granularity. HTA-OCR performs the best, closely followed by pure OCR, and both are much better than OpenMP. Fewer tiles means lower parallelism, thus, the overall scalability is worse than the previous case. A larger problem size is used in Figure 5.7c and 5.7d with different tiling, and similar results are observed, except that in both cases the task granularity is large enough to amortize HTA-OCR's task generation overhead.

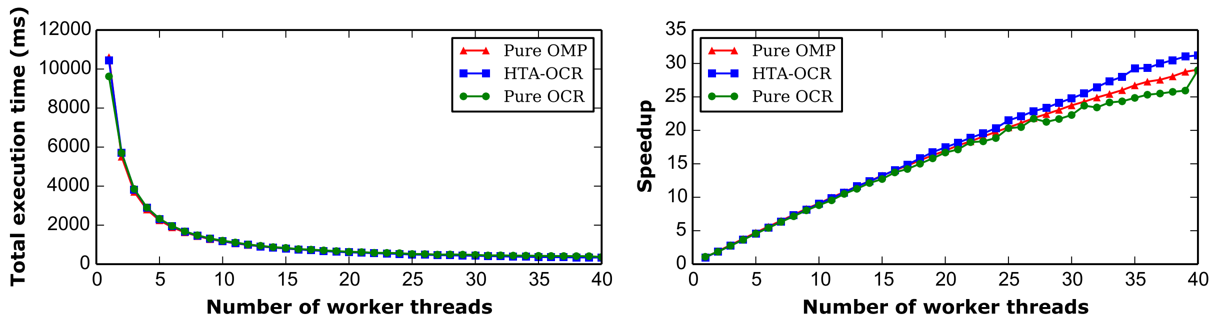
The experiments show that, for this problem, the OCR's task execution model indeed exploits the chances to overlap tasks better than OpenMP does. In the case of pure OCR, this involves programmer's effort to explicitly write code to construct the dataflow task dependence graph. But by using HTA-OCR, the programmer can write code with simplicity similar to that of the OpenMP version while getting the benefits of using the OCR runtime system.



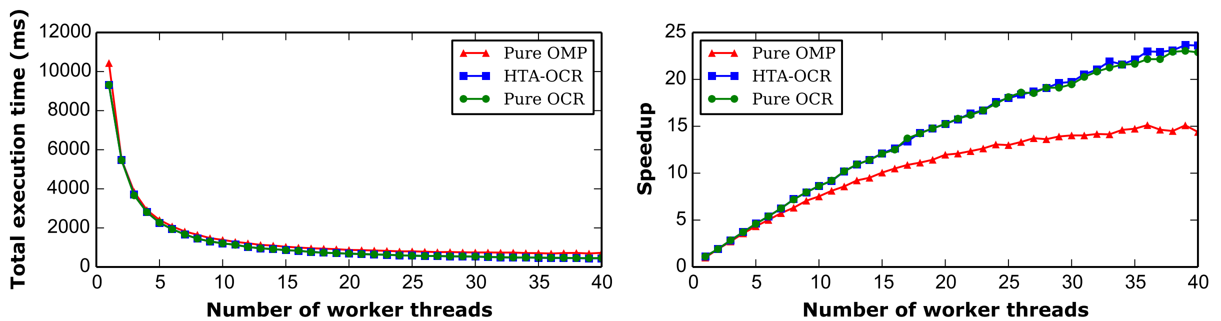
(a) 3200×3200 elements, 32×32 tiles, 100×100 elements per tile



(b) 3200×3200 elements, 16×16 tiles, 200×200 elements per tile



(c) 6400×6400 elements, 32×32 tiles, 200×200 elements per tile



(d) 6400×6400 elements, 16×16 tiles, 400×400 elements per tile

Figure 5.7: Dense Cholesky factorization results

5.4 Tiled Sparse LU Factorization

LU factorization converts an input matrix A into the product of a lower triangular matrix L and an upper triangular matrix U. Compared with Cholesky factorization, this method is not limited to Hermitian positive-definite matrices. Here, we consider a tiled version of the algorithm.

Our HTA-OCR tiled sparse LU factorization implementation is adapted from the source code used in the Teraflux project [14]. A code segment (Listing 5.7) generates the input matrix before LU factorization starts. It iterates over the blocks of the matrix, and follows pre-defined rules to decide whether a block is empty or not. If a block is non-empty, it allocates memory space for it. The code segment generates input matrices of the same sparsity pattern every time for the same tiling configuration.

```
/* NB: number of blocks along a dimension */
for (ii = 0; ii < NB; ii++)
  for (jj = 0; jj < NB; jj++) {
    null_entry = FALSE;
    if ((ii < jj) && (ii % 3 != 0)) null_entry = TRUE;
    if ((ii > jj) && (jj % 3 != 0)) null_entry = TRUE;
    if (ii % 2 == 1) null_entry = TRUE;
    if (jj % 2 == 1) null_entry = TRUE;
    if (ii == jj) null_entry = FALSE;
    if (null_entry == FALSE) {
      A[ii][jj] = (ELEM *)malloc(BSIZE * BSIZE * sizeof(ELEM));
      if (A[ii][jj] == NULL) {
        printf("Out of memory\n");
        exit(1);
      }
    } else
      A[ii][jj] = NULL;
  }
```

Listing 5.7: Code to generate parallel tiled sparse LU sparsity pattern

The HTA-OCR program code of the core of the computation is shown in Listing 5.8. An outer `kk` loop contains four steps:

1. `DIAG` factors the diagonal tile `(kk,kk)` sequentially into the lower triangular part (`A[kk].lt`) and the upper triangular part (`A[kk].ut`).

```
/* hA: The tiled sparse matrix to factor
    A: A helper 2D array for determining whether a block is empty
   NB: The number of blocks in a single dimension                  */

for (kk=0; kk<NB; kk++) {
  Tuple tkk = Tuple_create(2, kk, kk);
  /* (1) Diagonal tile */
  HTA_map(DIAG, ARGS(LHS(0, HTA_pick_one_tile(hA, &tkk))));
  /* (2) Row kk tiles */
  for (jj=kk+1; jj<NB; jj++) {
    if (A[kk][jj] != NULL) {
      Tuple tkj = Tuple_create(2, kk, jj);
      HTA_map(ROW_UPDATE, ARGS(LHS(0, HTA_pick_one_tile(hA, &tkj)),
                               RHS(0, HTA_pick_one_tile(hA, &tkk))));
    }
  }
  /* (3) Column kk tiles */
  for (ii=kk+1; ii<NB; ii++) {
    if (A[ii][kk] != NULL) {
      Tuple tik = Tuple_create(2, ii, kk);
      HTA_map(COL_UPDATE, ARGS(LHS(0, HTA_pick_one_tile(hA, &tik)),
                               RHS(0, HTA_pick_one_tile(hA, &tkk))));
    }
  }
  /* (4) Submatrix updates */
  for(ii=kk+1; ii<NB; ii++) {
    if (A[ii][kk] != NULL) {
      Tuple tik = Tuple_create(2, ii, kk);
      for (jj=kk+1; jj<NB; jj++) {
        if (A[kk][jj] != NULL) {
          Tuple tkj = Tuple_create(2, kk, jj);
          Tuple tij = Tuple_create(2, ii, jj);

          /* A[ii][jj] will be generated by this operation */
          if (A[ii][jj]==NULL)
            A[ii][jj] = NON_NULL;

          HTA_map(SUBMAT_UPDATE,
                  ARGS(LHS(0, HTA_pick_one_tile(hA, &tij)),
                       RHS(0, HTA_pick_one_tile(hA, &tik),
                              HTA_pick_one_tile(hA, &tkj))));
        }
      }
    }
  }
}
```

Listing 5.8: Parallel tiled sparse LU factorization in HTA-OCR

2. `ROW_UPDATE` solves `X` for the equation `A[kk][jj]=A[kk].lt*X` for each of the row `kk` tiles.

3. `COL_UPDATE` solves `X` for for the equation `A[ii][kk]=X*A[kk].ut` for each of the column

`kk` tiles.

4. `COL_UPDATE` update each of the tiles in the submatrix

   `A[ii][jj]-=A[ii][kk]*A[kk][jj]`.

The operations on the tiles within a step are fully independent, but there are data dependences between the steps. There are also dependences across iterations of the loop `kk`.

The data dependences form a complex task graph, as shown in Figure 5.8. The tasks are marked with the type of the operation and the index of their output tile. Similar to the Cholesky factorization in Section 5.3, the parallelism grows rapidly initially and diminishes with the progress of the execution. Notice that the task graph is drawn assuming all tiles are dense. When there are empty tiles, some of the nodes and dependence edges will be removed, resulting in a subgraph of the task graph.

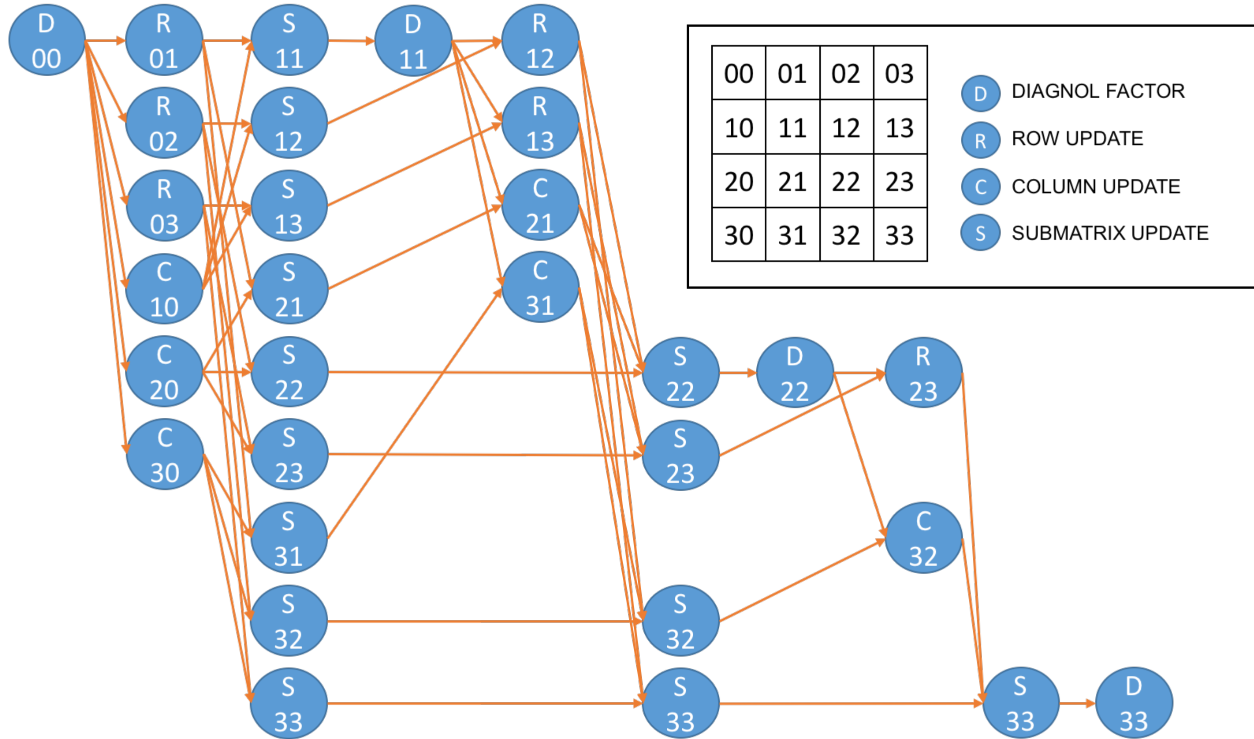
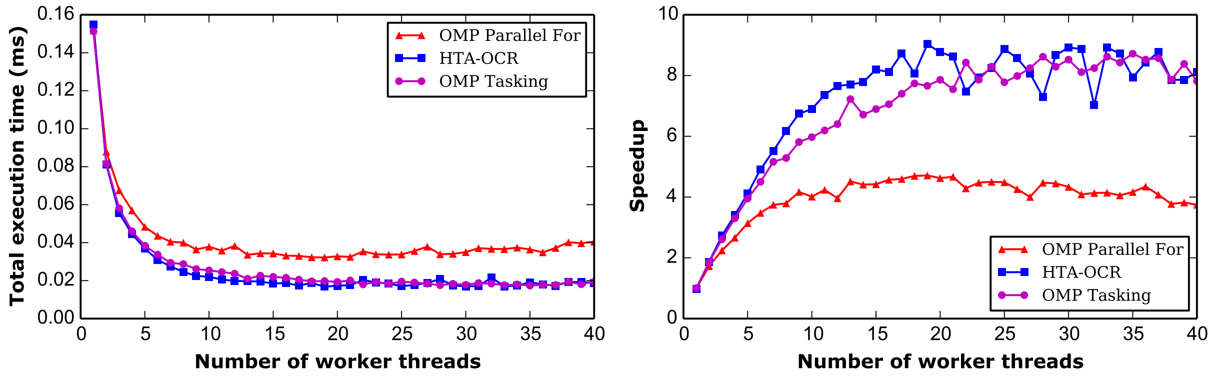


Figure 5.8: Sparse LU task graph with input HTA of 4×4 tiles

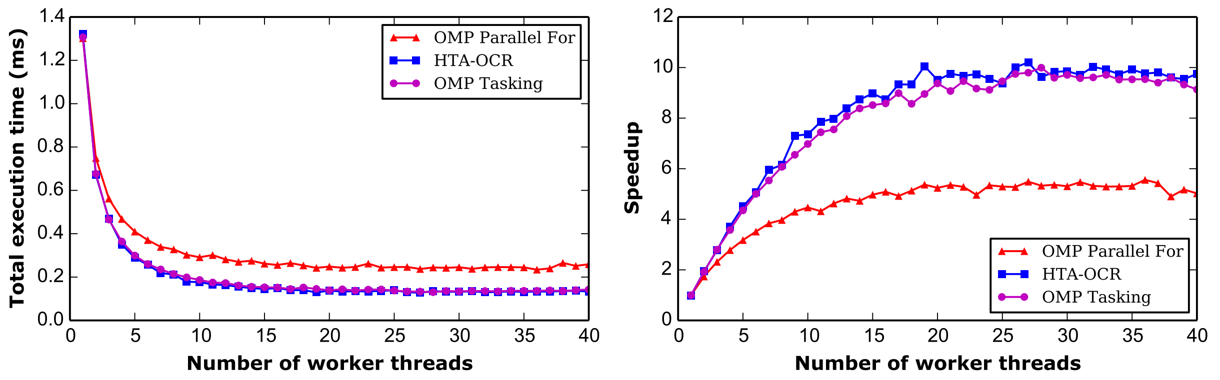
The dataflow task graph is even more complicated than Cholesky factorization for the

same tiling. Bulk synchronous execution is not desired, because it not only stops the computations of different steps but also the computations across the outer `kk` loop iterations from overlapping. This parallelism is automatically exploited in HTA-OCR by the underlying OCR runtime system because the parallel constructs do not enforce implicit barriers and the runtime system can schedule tasks as soon as their input data dependences are satisfied. Besides comparing to OpenMP using parallel for loops, we also compare to OpenMP Tasking, which incorporates ways to program data dependent tasks [8, 16]. Both the OpenMP versions used in our experiments are also adapted from the OMPSS version from the Teraflux project.

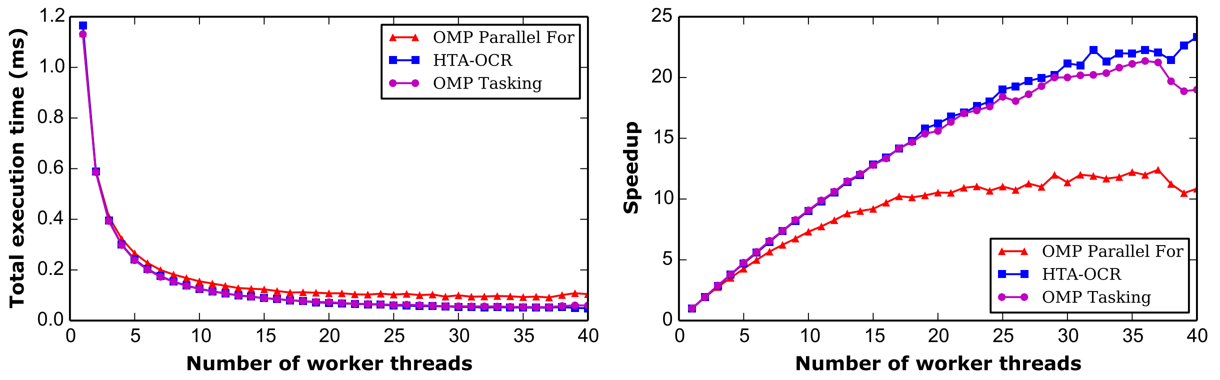
Figure 5.12 shows the performance results of the three versions. In 5.9a, an input matrix of 1600×1600 is used and divided into 16×16 tiles. As can be seen from the results, the HTA-OCR version performs comparably to the OpenMP Tasking version, and both can achieve better speedup than using OpenMP parallel for loops. This confirms our assumption that dataflow task execution model can deliver greater performance when the application has lots of asynchronous tasks. When we increase the task granularity by using the same tiling with larger tiles as shown in 5.9b, the maximum achievable speedup increases for all three versions, but not by a lot, because the tiling 16×16 does not expose enough parallelism in the computations. We use four times more tiles in 5.9c, and the results improve due to increased parallelism.



(a) 1600×1600 elements, 16×16 tiles, 100×100 elements per tile



(b) 3200×3200 elements, 16×16 tiles, 200×200 elements per tile



(c) 3200×3200 elements, 32×32 tiles, 100×100 elements per tile

Figure 5.9: Sparse LU factorization results

5.5 AMG Microkernel

AMG microkernel is one of the Coral benchmarks [29]. It has three kernels, and each of them is executed 500 times. Matvec computes sparse matrix-dense vector multiplication. Relax is algebraic multigrid mesh relaxation. Axpy computes $\alpha x + y$ where α is a scalar, x and y are vectors. We only evaluate HTA-OCR for the first two kernels, because Axpy uses dense vector and is embarrassingly parallel with workload easily balanced and it will show similar characteristics to benchmark programs that we have already discussed.

In the experiments, we measured the total time spent in executing all 500 iterations. We used a fixed problem size in the experiments: the sparse matrices are of dimension $10^6 \times 10^6$ and the dense vectors each has 10^6 elements. Notice that for both Matvec and Relax kernels, the input sparse matrix is generated once and used repeatedly in all 500 iterations. Thus, the non-zero patterns in the generated matrices do not change.

5.5.1 Matvec Kernel

For the Matvec kernel, the OpenMP implementation from the Coral website parallelizes the for loop that iterates over the rows of the sparse matrix using static scheduling. Hence, all threads get nearly equal number of rows as input. But since the matrix is sparse, the workload is most likely uneven, and the total execution time depends on the thread that gets the most non-zero elements in its set of rows. Also, because of the implicit barriers in OpenMP, when one matvec is not completely finished, the next one cannot start. In the HTA-OCR version, we take the generated sparse matrix and convert it into a sparse HTA. The rows are also split evenly among tiles. There are no barriers, but since the output vector is reused, the subtasks across iterations form dependence chains.

In Figure 5.10, it can be seen that HTA-OCR has better results. From the execution trace, we found that the average subtask execution time is way less in HTA-OCR than in

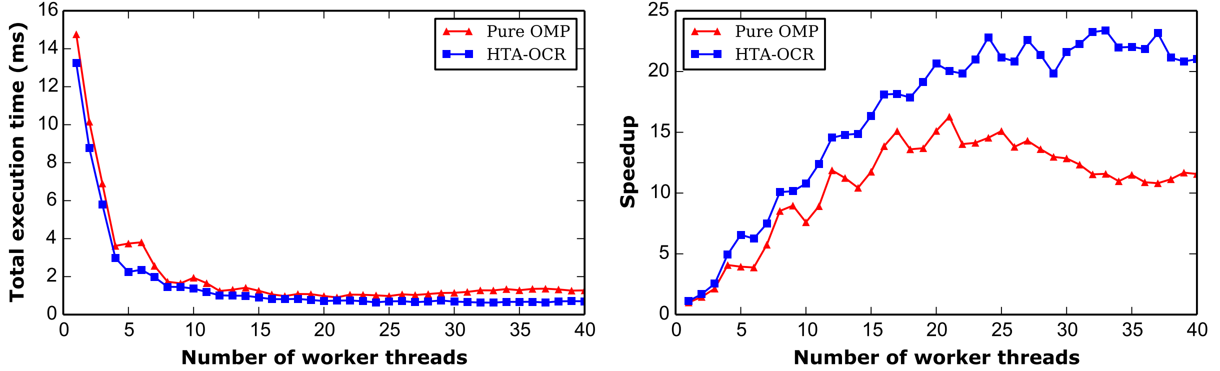


Figure 5.10: Matvec kernel results

OpenMP. We also profiled the cache misses and saw that HTA-OCR had less cache miss rate. The execution model difference does not seem to play a role here, because the tasks form dependence chains and there is not much asynchrony. We believe the different memory layouts in the sparse matrices of the two versions affect the average task execution time. In the OpenMP version, the generated sparse matrix (Compressed Sparse Row (CSR) format) is not tiled. But in HTA-OCR, the sparse matrix is converted into a sparse HTA by copying the data from the CSR matrix into an array of CSR tiles. This difference contributes to the much better performance of the HTA-OCR implementation.

5.5.2 Relax Kernel

The Relax kernel is an implementation of Successive Over Relaxation method. Its computation is shown in Algorithm 1. The parallelization is similar to that of Matvec and the rows of the sparse matrix are split evenly. In OpenMP, `#pragma omp for` is added to both the first copy loop (Line 1 - 3) and the second for loop (Line 4 - 10). For HTA-OCR, a map operation performs the copying and a subsequent map operation generates the `RELAX` tasks that have all-to-all dependences to the copy tasks.

We found that the HTA-OCR version does not perform well for this kernel as shown in

Algorithm 1: Relax Kernel Computation

Data: A is a matrix of size $n \times n$. f, u are both vectors of size n. tmp is also a vector used to hold the original value of u.

```
1  for i = 0 to n − 1 do
2  |  tmp_i = u_i;
3  end
4  for i = 0 to n − 1 do
5  |  res = f_i;
6  |  for j = i + 1 to n − 1 do
7  |  |  res -= A_ij × tmp_j;
8  |  end
9  |  u_i = res / A_ii;
10 end
```

Figure 5.11. The execution trace shows that there is a barrier-like behavior due to all-to-all dependences, and the `RELAX` subtasks in HTA-OCR take longer in average than OpenMP. We suspect this is a data locality issue and we use Papi [31] library to insert codes to read hardware cache miss event counters. We measured the cache misses rate of using 16 and 32 therads, as shown in Table 5.1. Surprisingly, not only the data cache miss rates are higher for the HTA-OCR version, the L1 I-cache miss rate are significantly higher too. This might be because when the OCR worker threads switch tasks, they run some OCR runtime routines and replace content from the L1 I-cache. And the higher miss rate in the data cache could be a result of bad runtime scheduling decisions causing the subtasks to be assigned to where the required data does not already exist in the cache.

Thread Count	**Version**	**L1 I-Cache Accesses**	**L1 I-Cache Misses (Rate)**	**L2 Total Cache Accesses**	**L2 Total Cache Misses (Rate)**
16	OpenMP	286,563,331	18,111 (0.006%)	33,542,300	4,680,982 (13.96%)
	HTA-OCR	267,827,293	45,374 (0.017%)	32,765,762	6,101,246 (18.62%)
32	OpenMP	145,291,535	16,647 (0.011%)	17,076,590	2,220,049 (13.00%)
	HTA-OCR	133,942,827	45,869 (0.034%)	16,887,818	2,749,603 (16.28%)

Table 5.1: PAPI hardware event counter report of AMG Microkernel Relax kernel execution

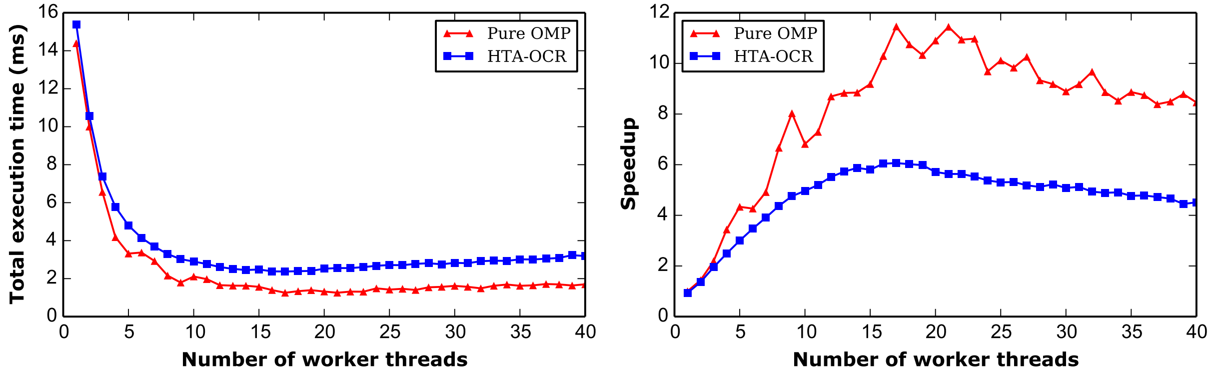


Figure 5.11: Relax kernel results

5.6 Parallel Mergesort

Mergesort is one of the most influential sorting algorithm. It uses a divide-and-conquer strategy. When the sub-problems have been sorted, merge steps are performed to get the solution to the original problem. The algorithm can be implemented by using either a recursive top-down approach or an iterative bottom-up approach. Intuitively, the parallelism comes from solving sub-problems in parallel independently[2]. We implement parallel mergesort using the bottom-up approach.

Listing 5.9 is the HTA-OCR mergesort. It is different from the one presented in Section 4.1 because the number of chunks is dynamic. For simplicity, we assume the number of chunks is a power of two. We use an HTA of multiple levels to store the data. The height of the HTA is computed in Line 4 by taking `LOG2` of the number of chunks. In Line 6 - 7, a regular array is allocated and initialized with unsorted integers. Next, an array of `Tuples` are filled with the desired tiling which has 2 tiles for each node in the HTA hierarchy except for the leaves. In Line 16 - 19, an HTA `hA` is created by using `HTA_create_impl()`, which allows creating a shell of HTA on the regular array `A` with the dynamically created `tiling`.

The major computation is between Line 22 - 27. First, an `HTA_map()` call applies `QSORT`

[2] A more sophisticated algorithm also parallelizes the merge step to get even more parallelism.

```
int CHUNK_SIZE   = /* Initialize for appropriate task granularity */
int num_elements = /* The size of the array */
int num_chunks   = num_elements / CHUNK_SIZE; /* Assume power of 2 */
int height       = LOG2(num_chunks);

double *A = malloc( ... );
/* ... Initialize array A ...*/

/* Create tiling dynamically */
Tuple tiling[height];
for(int i = 0; i < height; i++) {
  tiling[i] = Tuple_create(1, 2);
  tiling[i].height = height - i;
}
/* Create HTA hA using the content of array A */
Tuple flat_size = Tuple_create(1, num_elements);
/* ... Initialize dist ...*/
HTA *hA = HTA_create_impl(A, 1, height+1, &flat_size,
                          ORDER_ROW, &dist, HTA_SCALAR_TYPE_DOUBLE,
                          height, tiling);
/* Quick sort the leaves individually */
HTA_map(QSORT, ARGS(LHS(height, hA)));
/* Bottom up merge */
while(height > 0) {
  height--;
  HTA_map(MERGE, ARGS(LHS(height, hA)));
}
```

Listing 5.9: Parallel mergesort in HTA-OCR

to the leaf level tiles of `hA`. For each leaf tile, a parallel subtask is created to perform quick sort on its data. Then, in each iteration of the while loop, `HTA_map()` is called to apply `MERGE` to a single level, starting from one level above the leaf to the root level. Each `MERGE` subtask gets either an intermediate node or the root node as its input. Either way, it merges the data of its two children nodes. The task graph formed dynamically is similar to the one shown in Figure 4.2. As a comparison, the OpenMP implementation is shown in Listing 5.10.

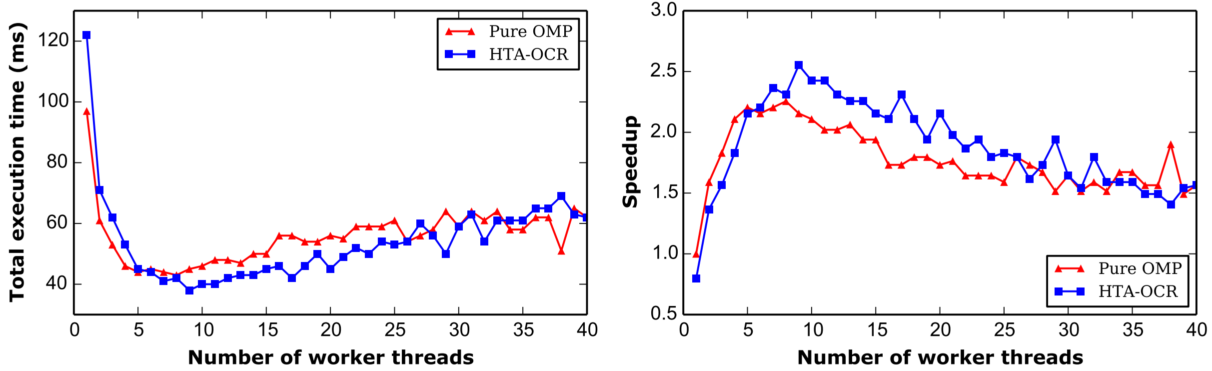
We conducted experiments to compare the strong scaling (fixed problem size) of the HTA-OCR version with its OpenMP counterpart. In the first experiment, the input array, which is not sorted, has 2^{20} double precision floating numbers divided into 2^{15} elements per leaf tile and there are 2^5 leaf tiles in total. The execution time and speedup with respect to the amount of OCR worker threads are shown in Figure 5.12a.

```
1 int CHUNK_SIZE   = /* Initialize for appropriate task granularity */
2 int num_elements = /* Initialize to size of the array */
3 int num_chunks   = num_elements / CHUNK_SIZE; /* Assume power of 2 */
4
5 #pragma omp parallel for schedule(runtime)
6 for(long int i = 0; i < num_chunks; i++) {
7   QSORT(&A[i*CHUNK_SIZE], CHUNK_SIZE);
8 }
9
10 int level_chunk_size = CHUNK_SIZE;
11 int num_merges = num_chunks >> 1;
12
13 while(num_merges > 0) {
14   #pragma omp parallel for schedule(runtime)
15   for(int i = 0; i < num_merges; i++) {
16     MERGE(&A[2*i*level_chunk_size], &A[(2*i+1)*level_chunk_size],
         level_chunk_size);
17   }
18   level_chunk_size <<= 1;
19   num_merges >>= 1;
20 }
```

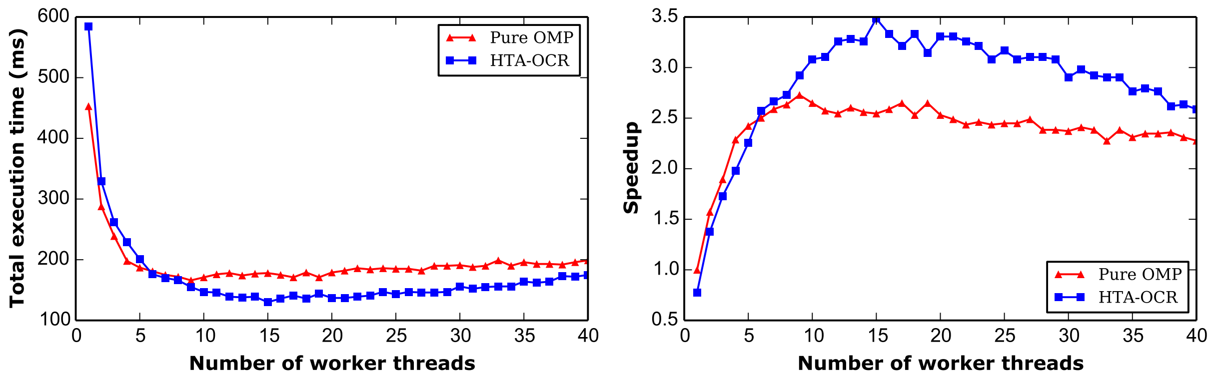
Listing 5.10: Parallel mergesort in OpenMP

The strong scaling results are not very good for either implementation. This is because the algorithm has the most parallelism in the beginning but the parallelism quickly decreases. Increasing the number of worker threads adds overhead but does not help when there is not enough parallelism.

When the number of threads used is small, the HTA-OCR version performs slightly worse than the OpenMP counterpart. This is due to the higher overhead of the OCR tasks and the added overhead of the HTA library layer. However, it catches up when the thread count is above 6, and achieves the best speedup (2.55×) when thread count is 9. This is because the OCR runtime has the advantage of dynamic scheduling while the OpenMP implementation relies on barriers. The results when the array is 4 times larger with the same leaf tile size and more leaf tiles are shown in Figure 5.12b. Both HTA-OCR and OpenMP have better speedup results than in the previous experiment. The maximum speedup (3.48×) is again achieved by the HTA-OCR version, and it is higher than the previous configuration because of the increased parallelism.



(a) 2^{20} elements, 2^5 leaf tiles



(b) 2^{22} elements, 2^7 leaf tiles

Figure 5.12: Parallel mergesort results

5.7 K-means

K-means is a classic unsupervised machine learning algorithm. For a given set of data samples, each has a set of numerical values as their features, the K-means algorithm finds the centroids of data clusters. Suppose K centroids are desired as the result, the iterative algorithm starts by guessing the K centroids, and then classifies the samples with these guesses by choosing for each sample the closest centroid. Next, with all samples classified, there are now K groups of samples. The centroids of groups are then updated by taking the average of the samples in each of them. The newly computed centroids are used to classify the data samples in the next iteration. A convergence check is performed in each iteration

to see if the centroids stop changing values.

We acquired the baseline OpenMP code from the Rodinia benchmarks [13]. The implementation parallelizes the classification of the data samples using a parallel for loop. The data samples are divided into even-sized chunks using static scheduling. For each thread, it also uses extra storage to keep the sum of the samples while doing the classification, and to track if the assigned group of any sample changes, as the convergence test. Once the parallel for loop finishes, the partial sums from all threads are summed up and the average is taken to get the new centroids in the sequential section. The outer iteration loop tests whether the result converges or the maximum number of iterations is reached.

The HTA-OCR version also divides the data samples array into even-sized tiles. It parallelizes the classification step using `HTA_map()`. The partial sums of the samples and the local convergence of each chunk are also computed in the same map operation. Three reductions are required here: one for the partial sums of the samples in each group, one for the counts of the samples in each group, and one for merging the convergence test results. Finally, another map operation is needed to update the centroids.

We used the dataset `kdd_cup` that comes with the Rodinia K-means benchmark, which has 494020 samples. Initially, the OpenMP K-means program from Rodinia does not have any speedup using multiple threads. After some investigation, we found there is a performance bug due to the false sharing of the extra storage allocated for threads to count the number of samples in each group in parallel. We added padding to the rows of the arrays and it solved the performance bug. In the results shown in Figure 5.13, we see that the OpenMP version performs much better than HTA-OCR. After studying the HTA-OCR execution trace, we determined that the inferior performance is because of the three reductions. These reductions cause three split-phase continuations and create a large gap between the classification tasks. In contrast, the OpenMP reductions are much faster and the OpenMP version scales perfectly for K-means algorithm up till 37 threads.

A way to improve the problem in HTA-OCR is to merge the three split-phase continuations into a single one. In the K-means algorithm, there are no data dependences among the three reductions. Hence, there is no need to split-phase continue until immediately before the use of the reduction results. Although our current implementations for reductions are *synchronous*, it is possible to implement *asynchronous* reductions, and an extra operation to retrieve the results from different reductions at once, so that the split-phase continuations can be delayed and the number of them can be curtailed.

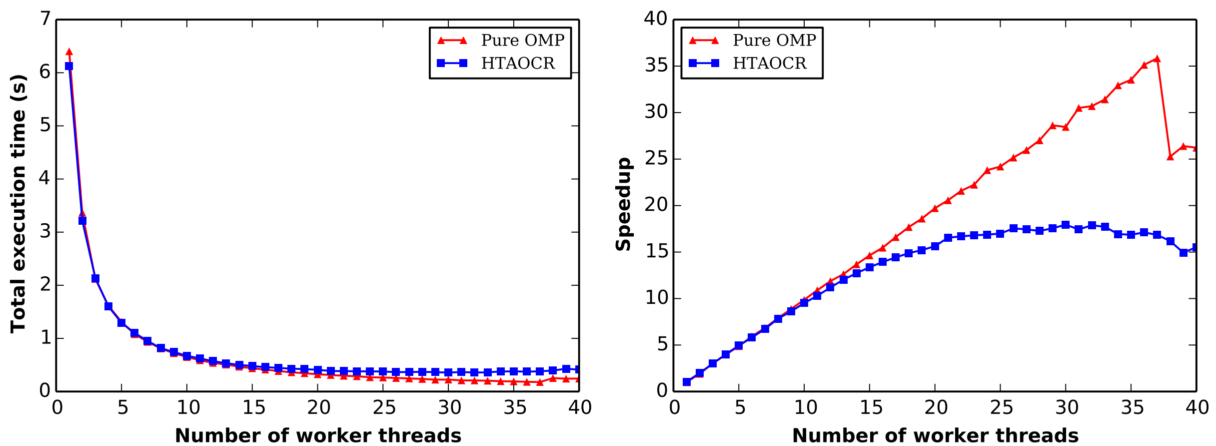


Figure 5.13: Kmeans results

5.8 Parallel Breadth First Search

Breadth first search (BFS) in a graph starts from a single node, and then gradually expands to the neighbors of the nodes that have been searched. The level-synchronous algorithm is often used. It is called level-synchronous because the algorithm uses a worklist to store the active nodes whose neighbors needs to be searched, and the active nodes in the worklist of an iteration has the same minimal distance (i.e. at the same level) from the source node. In this experiment, we use the reference OpenMP implementation from the Graph 500 [34] challenge as the base to build an HTA-OCR version.

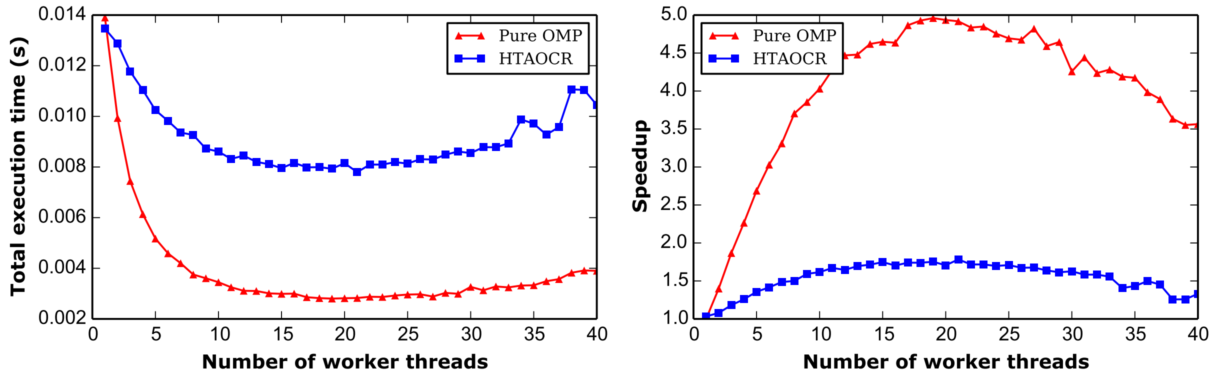
The OpenMP version in Graph 500 is a level-synchronous algorithm. It uses an array `vlist` as the worklist of the active nodes. At the beginning, only the source vertex is in the list, and the first iteration of the BFS loop pops the source vertex and searches its neighbors. The neighbors that have a distance of 1 from the source vertex are added to the worklist. In the next iteration, all vertices in the worklist are popped, and their neighbors are searched in parallel. It is possible that more than one active nodes have the same neighbors. The OpenMP implementation uses an atomic compare-and-swap operation to ensure that the same neighbor is picked only once at each level.

In the HTA-OCR version, we also use the same algorithm but with some changes to make it work with HTA-OCR. We keep the `vlist` array, and at each level, we use `HTA_create_shell()` function to construct an HTA containing the active nodes of the level. We tile the active node list using a user specified number `NP`. When the number of active nodes is fewer than `NP`, we let each tile contain only one active node. A map operation applies `BFS` on the tiles to perform the search of neighbors in parallel. It also builds a local new active node list, and counts the number of newly added nodes. A prefix scan is performed on the local counts, and then the results are used to merge the local active node lists into one which is then appended to the `vlist` array. The results of the scan are also used to

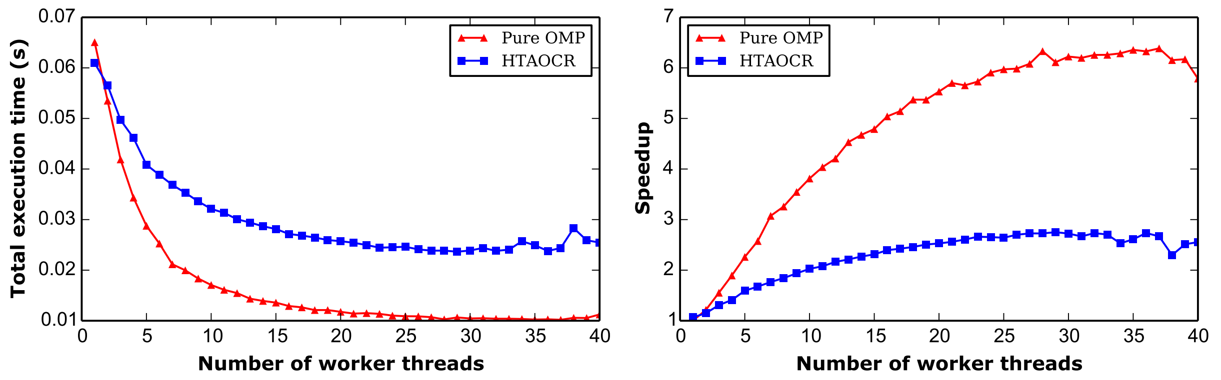
update the range of the active nodes in the `vlist`. While we try to keep the algorithm the same as OpenMP, we have to use the atomic compare-and-swap operation in the subtasks. This breaks the OCR model, since the subtasks synchronize with each other in the middle of their executions and not through input dependence slots. But for a single node machine, this algorithm works fine and is the closest to the OpenMP version.

The benchmark allows adjusting the problem size by specifying the *SCALE* and the *edge factor*. SCALE is the logarithm base two of the number of total vertices in the graph, and the edge factor is the ratio of the total number of edges to the number of vertices. In the configurations of our experiments, we varied the SCALE but kept the edge factor the same. The smallest size we used has 2^{16} vertices and the largest one has 2^{20} vertices. We kept the other default configurations of the benchmark.

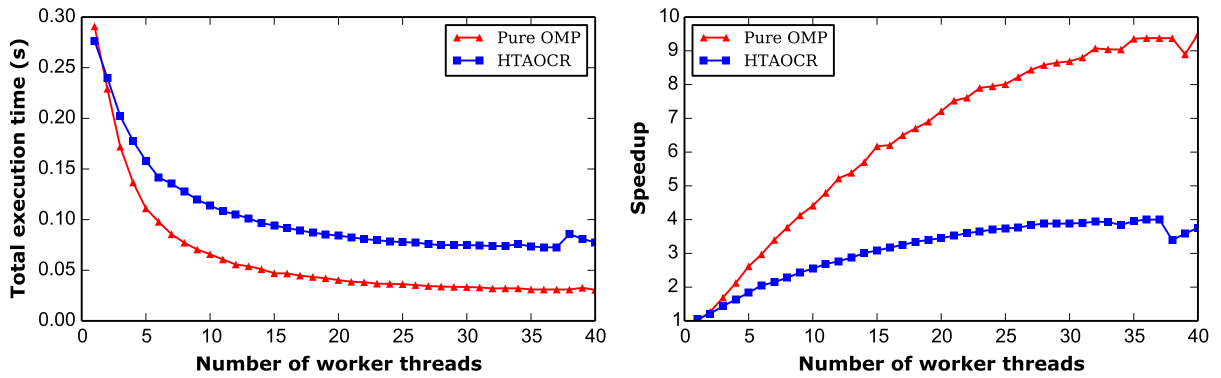
The performance results of parallel BFS are the worst for HTA-OCR of all the benchmarks we evaluated in this chapter. For the default configuration in Figure 5.14a, the speedup never reached 2×. When we increased the problem size in 5.14b and 5.14c, while the results did improve, they are still disappointing. We examined the execution trace and found a few problems. First, the HTA-OCR implementation needs to read back the prefix-scan result from an HTA at each level, causing a split-phase continuation. This is much slower compared to using global synchronization barriers and performing memory reads after the barrier. Second, the workload at each level varies a lot, and it does not solely depend on the number of active nodes. It is possible that there is a large number of active nodes but the number of the neighbors to search is small. This makes the task granularity very fine, and the task overhead in HTA-OCR prevents it from getting good performances.



(a) SCALE = 16, Edge factor = 16



(b) SCALE = 18, Edge factor = 16



(c) SCALE = 20, Edge factor = 16

Figure 5.14: Parallel BFS results

5.9 LULESH

Livermore Unstructured Lagrange Explicit Shock Hydrodynamics (LULESH) [24] is a proxy app designed by Lawrence Livermore National Lab. It has been studied in many research projects, and there have been attempts to port it to different programming models and advanced machines. The current version is LULESH 2.0. For this experiment, due to the lacking of a C implementation of LULESH 2.0 [28], we asked the author's help and got a sequential C implementation of LULESH 1.0, which we ported to OpenMP and HTA-OCR.

The LULESH algorithm uses a timestepping method to simulate 3-D shock hydrodynamics. In each timestep, a first phase of calculating nodal forces, accelerations, velocities, and positions is followed by a phase of calculating element quantities. Before starting the next timestep, hydro and courant constraints are calculated, and the delta time depends on the constraints. The source code of the OpenMP-C implementation of LULESH 1.0 is in a single file of around 3000 lines. There are 45 loops parallelized using `#pragma omp for` and there are two places where reductions are needed to compute the hydro and courant constraints.

Based on the OpenMP-C implementation, we ported it to HTA-OCR. Our strategy for porting is to convert the arrays into HTAs. The user has to specify a fixed number `NP`, and all the arrays are partitioned into `NP` tiles. The OpenMP parallel for loops are converted into `HTA_map()` calls, and the original loop bodies are moved to form the operator functions. See Listing 5.11 and 5.12 for an example of the contrast between the two. As for the reductions, each is represented by a map operation to compute partial results in parallel followed by an HTA reduction.

```
static inline
void CalcAccelerationForNodes(Real_t *xdd, Real_t *ydd, Real_t *zdd,
                              Real_t *fx, Real_t *fy, Real_t *fz,
                              Real_t *nodalMass, Index_t numNode){
   #pragma omp parallel for firstprivate(numNode)
   for (Index_t i = 0; i < numNode; ++i) {
      xdd[i] = fx[i] / nodalMass[i];
      ydd[i] = fy[i] / nodalMass[i];
      zdd[i] = fz[i] / nodalMass[i];
   }
}
```

Listing 5.11: LULESH OpenMP code using parallel for loop

```
void
CALC_ACCELERATION_FOR_NODES(HTA** lhs, HTA** rhs, uint64_t* capture){
  Index_t numNode = lhs[0]->flat_size.values[0];
  Real_t *xdd = HTA_get_ptr_raw_data(lhs[0]);
  Real_t *ydd = HTA_get_ptr_raw_data(lhs[1]);
  Real_t *zdd = HTA_get_ptr_raw_data(lhs[2]);
  Real_t *fx = HTA_get_ptr_raw_data(rhs[0]);
  Real_t *fy = HTA_get_ptr_raw_data(rhs[1]);
  Real_t *fz = HTA_get_ptr_raw_data(rhs[2]);
  Real_t *nodalMass = HTA_get_ptr_raw_data(rhs[3]);
  for (Index_t i = 0; i < numNode; ++i) {
    xdd[i] = fx[i] / nodalMass[i];
    ydd[i] = fy[i] / nodalMass[i];
    zdd[i] = fz[i] / nodalMass[i];
  }
}
static inline
void CalcAccelerationForNodes(HTA *xdd, HTA *ydd, HTA *zdd,
                              HTA *fx, HTA *fy, HTA *fz,
                              HTA *nodalMass, Index_t numNode){
  HTA_map(CALC_ACCELERATION_FOR_NODES,
          ARGS(LHS(1, xdd, ydd, zdd),
               RHS(1, fx, fy, fz, nodalMass)));
}
```

Listing 5.12: LULESH HTA-OCR code using HTA_map()

Problem Size	**Iteration Count**
45^3	1495
70^3	1816
90^3	2026

Table 5.2: The configurations used in LULESH experiments

We conducted the experiments using three different problem sizes as shown in Table 5.2. When the problem size increases, not only the time spent for one iteration increases, it also takes more iterations to converge. For the results of the smallest problem size shown in Figure 5.16a, the OpenMP version scales fine but not great. The maximum speedup 7.62× can be observed using 25 threads. In contrast, HTA-OCR can match OpenMP only up till 6 threads. It cannot compete with OpenMP for larger thread counts, and it even has slow down when more than 38 worker threads are used. Obviously, for this problem size, the task granularity is too small, and the overhead of using HTA and OCR library is prominent. Following this thread of reasoning, we increase the problem size to see if it improves the scalability. We can see that it indeed improves the scalability in Figure 5.16b. While the scalability of both improves, HTA-OCR can match OpenMP performance up till 17 threads. In the largest problem size we use in Figure 5.16c, the HTA-OCR version has even better performance for a range of thread count. However, the results do not justify the use of HTA-OCR in LULESH, because the programmability of the two models are similar and using HTA-OCR does not bring significantly better performance results.

The first reason for HTA-OCR's disadvantage in LULESH is because of the algorithm used. Two things limit the subtask overlapping. First, the need of reductions for convergence test in each iteration causes split-phase continuation in every iteration. As we mentioned in Section 4.5, split-phase continuation stops subtasks from being generated, and can cause temporary work depletion. Because of this, subtasks of different timesteps cannot overlap. Second, in a few places there are all-to-all data dependences between consecutive map operations, causing barrier-like behaviors. Compared with the optimized OpenMP barriers, the overhead of dynamically creating the subgraph to represent all-to-all dependences and executing it is much larger.

The other problem is the task granularity control. We straightforwardly split the arrays into as many tiles as the amount of worker threads, and the tile count is equivalently the

parallelism that can be utilized. However, for some simpler operations, even though they do have this much parallelism, the tasks are too fine grained. We can see this phenomenon from the execution trace in Figure 5.15. Some operations have coarse-grained tasks, but many others are very fine-grained. If the programmer can predict the subtask workload of an application, it could be better if hints are given to the library to adaptively adjust the amount subtasks generated. The other possibility is to merge some map operations to both coarsen subtasks and reduce the amount of subtasks generated. This can be done either by the programmer explicitly, or by the compiler statically, although a sophisticated dependence analysis is required.

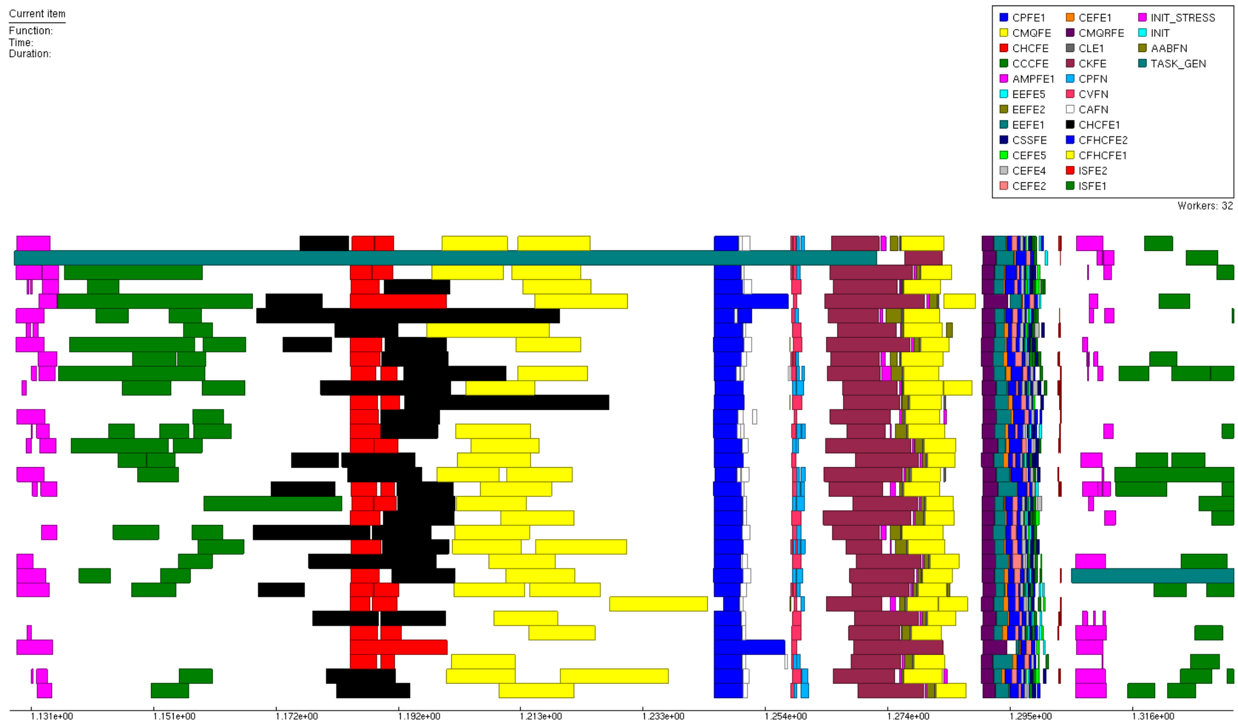
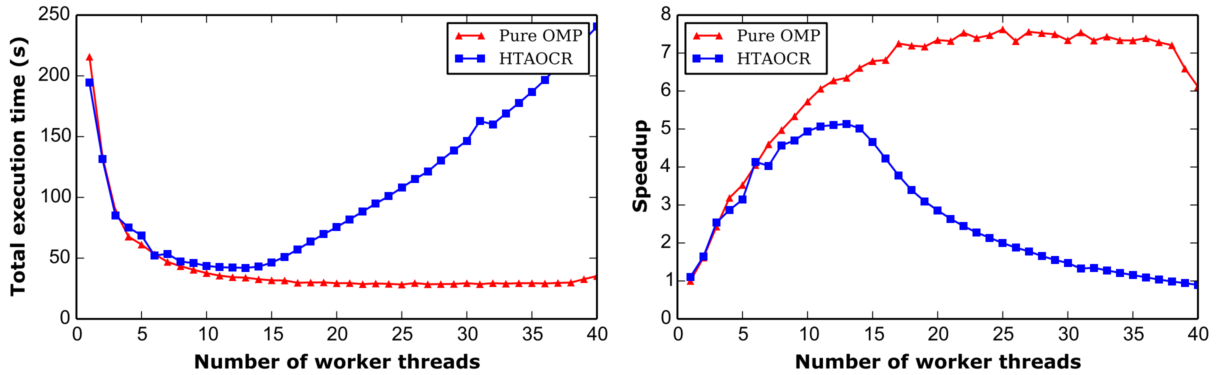
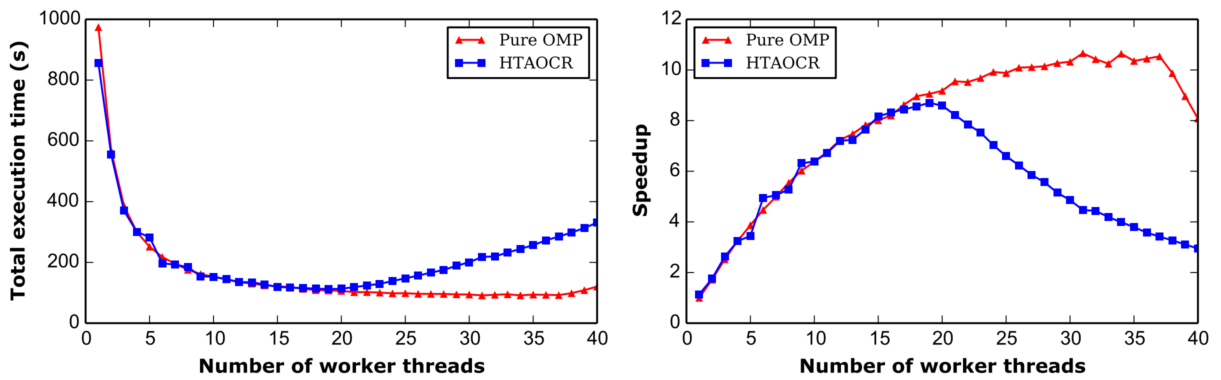


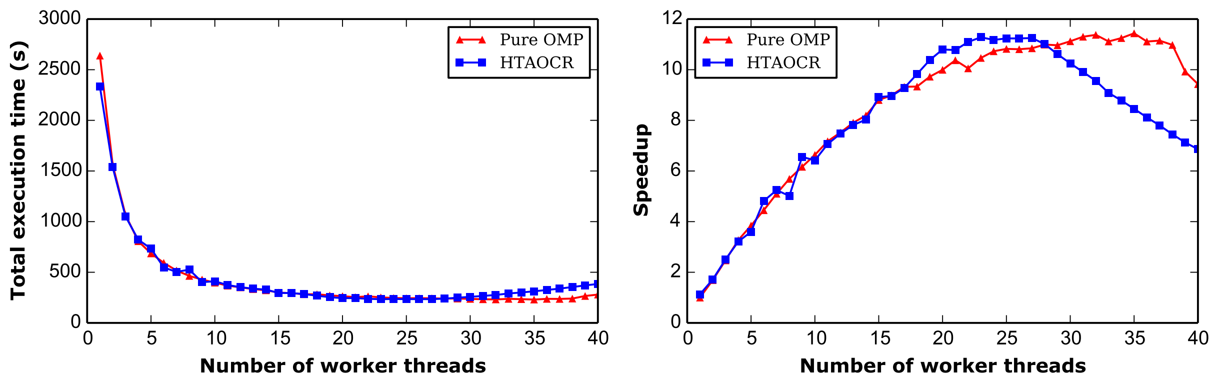
Figure 5.15: LULESH execution trace of a single timestep using 32 worker threads and edgeElem = 90



(a) edgeElems = 45



(b) edgeElems = 70



(c) edgeElems = 90

Figure 5.16: LULESH results

5.10 NAS Parallel Benchmarks

NAS (Numerical Aerodynamic Simulation) Parallel Benchmarks [4] are a set of programs created by NASA for evaluating the performance of parallel supercomputers. Although not full applications, they capture the behaviors of practical computational fluid dynamics applications. Thus, whether parallel programming systems and parallel machines can deliver good performance for them is a good indication of how well they will perform with real large-scale applications.

For our purpose of evaluating the HTA-OCR library, we implemented six of the benchmarks in HTA-OCR and compared them with the OpenMP implementations on a single node machine. We observe the strong scaling results of class C for every problem.

5.10.1 EP

EP stands for Embarrassingly Parallel. The program computes the Gaussian random deviates (X_k, Y_k) followed by a tabulating step to sum up the counts of the pairs that lie in specific square annulus. The sums $\sum_k X_k$ and $\sum_k Y_k$ are also computed.

Generally, the algorithm consists of three steps. First, it computes Guassian deviates (X_k, Y_k) fully independently. Second, there are the reductions to compute the counts of the pairs to be stored in table Q_l depending on the criteria given. Finally, there are the reductions to compute $\sum_k X_k$ and $\sum_k Y_k$. In OpenMP, the algorithm is implemented using a single parallel for loop to parallelize the computations of Gaussian deviates with reduction clause to sum up $\sum_k X_k$ and $\sum_k Y_k$. The entries of table Q_l are computed by using `atomic` addition inside of the parallel section.

Porting it to HTA-OCR is straightforward. First, a map operation takes input arrays and calculates Gaussian deviates, while also counting the pairs as partial Q_l results. Next, a reduction is performed to combine the partial Q_l results. Finally, two reductions are needed

to compute $\sum_k X_k$ and $\sum_k Y_k$ separately.

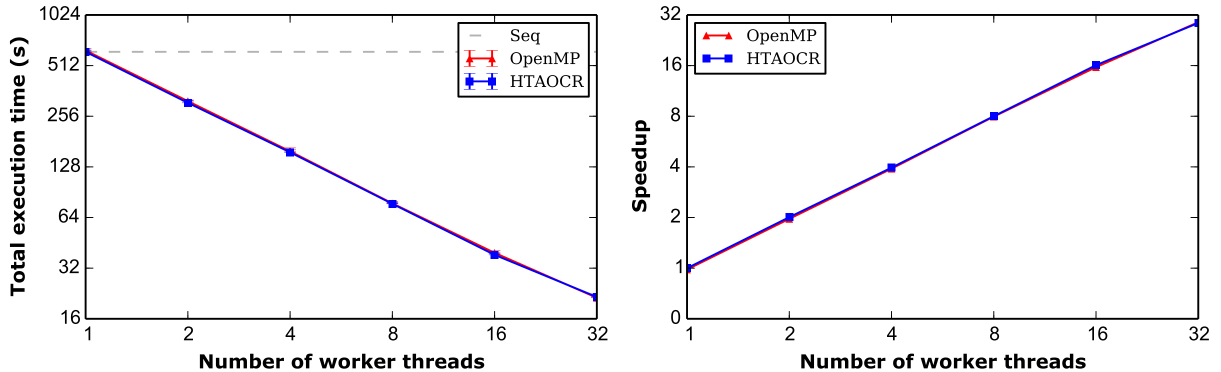


Figure 5.17: EP benchmark result

The results of both implementations are very close as we expected. The workload of the major computation of Gaussian deviates is perfectly balanced and is done fully independently. For the reductions at the end, HTA-OCR has more overhead since it has to generate separate tasks for them, but this part is not apparent because it is amortized by the major computation.

5.10.2 IS

Integer Sort (IS) sorts an integer array in parallel. The benchmark specification does not specify the algorithm to use, but only specifies the input and the desired outcome. It also specifies that only the computation of the ranks of the keys needs to be timed, but not the final rearrangement of the keys into the sorted array.

The OpenMP implementation uses a bucket sort algorithm for computing the ranks. Here we give a sketch of the algorithm:

1. In a parallel section, each thread is assigned a chunk of the unsorted key array. The threads then count the keys that fall in the buckets independently in a parallel for loop and establish counter values locally.

2. After the barrier of the previous step, each thread computes the range it has to fill in the global array of buckets by reading the counter values in the previous step.
3. Each thread sorts local keys into global array of buckets in a parallel for loop. The buckets are sorted after this step, but the keys within the buckets are unsorted.
4. In a dynamic schedule parallel for loop, each iteration the loop body determines the ranking of the keys in one bucket. The timed section ends after this step.

For the HTA-OCR version, we also use the bucket sort algorithm but with some changes for the reason that parallel tasks in HTA cannot randomly access the tiles they do not own. For this reason, the memory storage required is more than the OpenMP version.

1. The unsorted array is converted into an HTA of P (given by the user) tiles. An HTA of $B \times P$ buckets and an HTA of bucket size counters are also created.
2. `HTA_map()` operations are invoked to count the local bucket sizes and sort the keys into local buckets. Notice that there are P sets of buckets when this step is completed.
3. Now, decomposing the array of rankings, P tasks each ranks a range of keys by reading the results from the previous step.
4. Perform parallel scan to get the rankings of all keys.

The performance results of the two implementations are shown in Figure 5.18. Although a different algorithm is used in the HTA-OCR implementation, the performance results are close to OpenMP. In the 32-thread case, OpenMP achieved 26.63× speedup while HTA-OCR got 24.85× speedup.

5.10.3 FT

FT benchmark uses the 3-D fast-Fourier transform (FFT) to solve a partial differential equation. 3-D FFT is an important algorithm for many signal processing problems. The

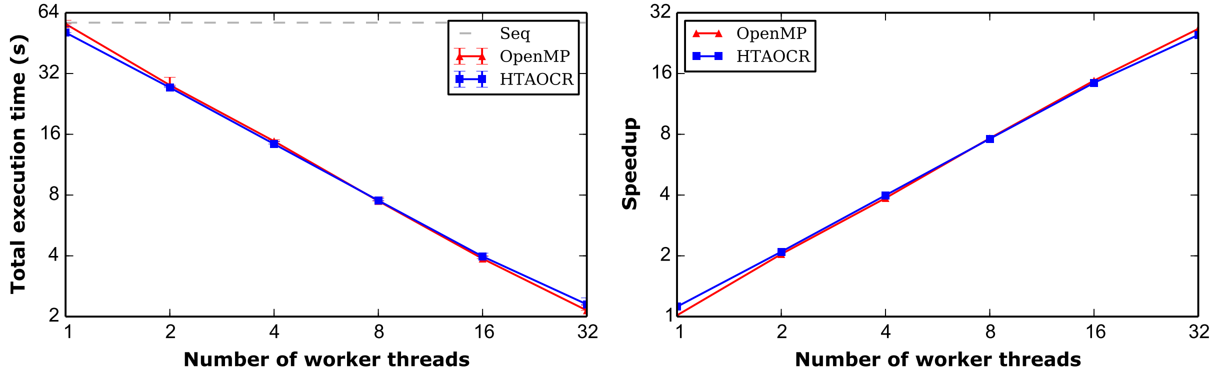


Figure 5.18: IS benchmark result

sketch of the program is shown below in Algorithm 2. The forward and inverse FFT are both performed with three consecutive 1-D FFTs in each of the three dimensions, only with different order.

Algorithm 2: FT algorithm

1 Initialization;
2 Forward FFT;
3 **for** $i = 0$ **to** N **do**
4   Evolve;
5   Inverse FFT;
6   Checksum;
7 **end**

The steps can all be parallelized using OpenMP parallel for loop. For each 3-D FFT, changing the dimension of applying 1-D FFT is done by varying order of the nested loops without having to actually transpose the 3-D array, and the implementation always parallelizes the outermost loop. In contrast, while most of the other steps in OpenMP can be converted to HTA map operations, 3-D FFT requires a transposition so that 1-D FFT can be applied correctly to all three dimensions. Since we are comparing peformance results on a single node, we take advantage of the shared memory space and create two-level HTAs that are transposed aliases of each other so that we can avoid actually transposing the 3-D array.

With this optimization, the HTA-OCR performance can be close to OpenMP, as shown in Figure 5.19.

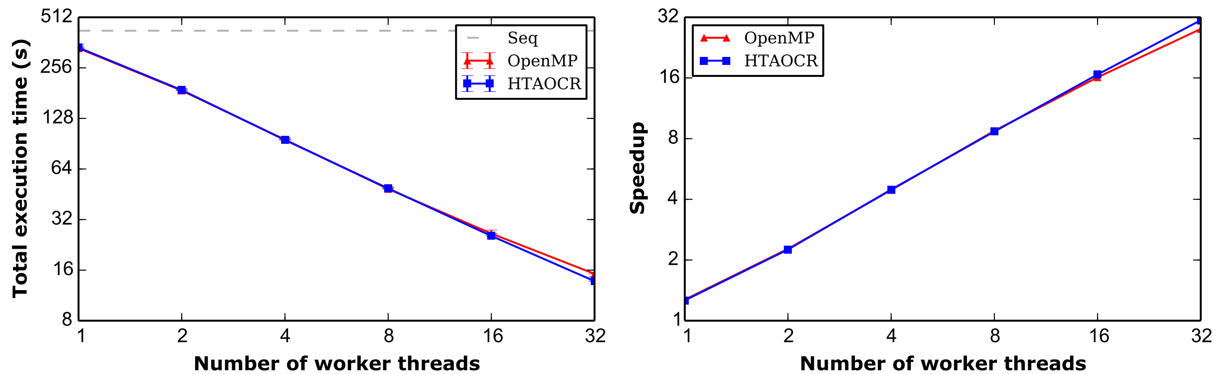


Figure 5.19: FT benchmark result

5.10.4 MG

MG benchmark solves a discrete Poisson problem by performing four iterations of V-cycle multigrid algorithm. In each V-cycle, there are two phases: the down-cycle and the up-cycle. In the down-cycle, an input 3-D grid is restricted to the coarsest level grid in a series of projection (`rprj3`) operations. Next, an approximate solution is calculated by applying `psinv` on the coarsest-level grid. The up-cycle then starts in a loop of interpolation (`interp`), residual computation (`resid`) and smoother application (`psinv`) iterations, till the resulting grid reaches the finest level.

The OpenMP parallelization is straightforward. In each of the operations mentioned, a `#pragma omp parallel for` is added to the outermost of the nested loops (i.e. 1-D decomposition). In contrast, the HTA-OCR version uses 3-D decomposition to partition the grid into tiles, we need to let the tiles contain ghost regions (i.e. copies of cells from neighboring tiles), since `psinv`, `rprj3` and `resid` are stencil computations. Only when tiles contain ghost regions can these operations be performed with `HTA_map()`. The ghost regions

also need to be updated when new output of these operations are computed. The data dependences due to ghost region updates are all-to-all dependences, and thus the amount of task overlapping is limited in the HTA-OCR version.

In HTA-OCR, the other change we have to make is for the computation on the coarser grids. When the finest grid has a small size, it is possible that the coarser grids cannot be tiled in the same way as the finest grid. In that case, the coarser grids are not tiled, and the operations performed on them becomes sequential. It is reasonable because using many parallel subtasks for a small grid and a small total workload results in the overhead dominating the cost.

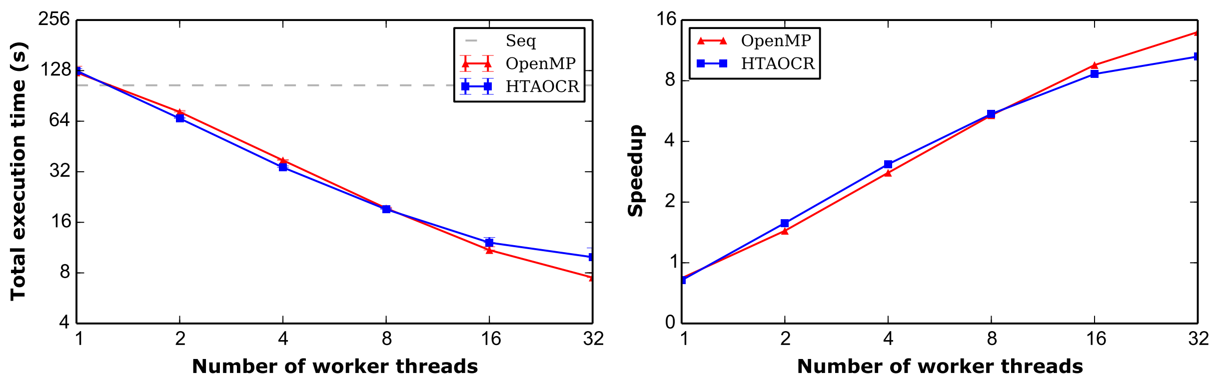


Figure 5.20: MG benchmark result

In Figure 5.20, we see that the HTA-OCR results are not as good as OpenMP for higher thread counts. Other than the all-to-all data dependences due to ghost region synchronizations in the program, we observe from the execution trace (Figure 5.21) that the when the program processes the coarser grids, the task granularity becomes very small and the task overhead is too much. OpenMP does not have the same problem because its global barrier synchronization is much faster. Since the problem is fix-sized and the algorithm does not grant flexibility in task overlapping, high thread count and task count are very disadvantageous for HTA-OCR.

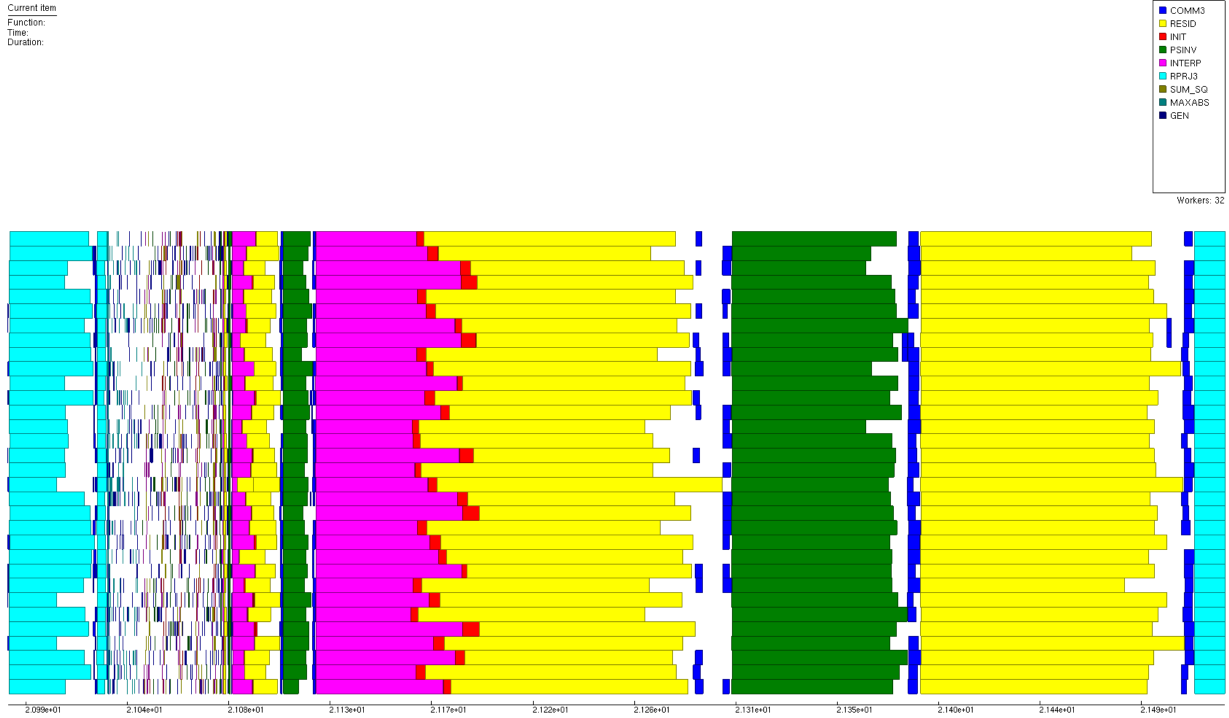


Figure 5.21: HTA-OCR MG V-cycle execution trace

5.10.5 CG

CG benchmark uses conjugate gradient method to solve an unstructured sparse linear system. The major computation is performed in a loop of 25 conjugate gradient iterations containing a sparse matrix-dense vector multiplication, vector inner product, and several *axpy* operations.

Both the OpenMP and HTA implementations parallelize the matrix-vector multiplication using the same methods as mentioned in Section 5.5.1. For the *axpy* operations, adding `#pragma omp for` in OpenMP is enough. In HTA-OCR, they are converted to `HTA_map()` operations. Inner products are calculated in OpenMP by using `reduction` clause in addition to `#pragma omp for`. As for HTA-OCR, a pair of `HTA_map()` and `HTA_FULL_REDUCE()` is needed.

The experimental results (Figure 5.22) show that although close, HTA-OCR does not perform as well as OpenMP. We think this is mainly because of the reductions causing

split-phase continuations, and, in each phase, there are not abundant asynchronous parallel subtasks for the worker threads to work on. The execution proceeds in a bulk synchronous manner due to operations depending on reduction results. HTA-OCR's overhead in task management makes it difficult to compete with the more sophisticated OpenMP implementation when the execution is bulk-synchronous.

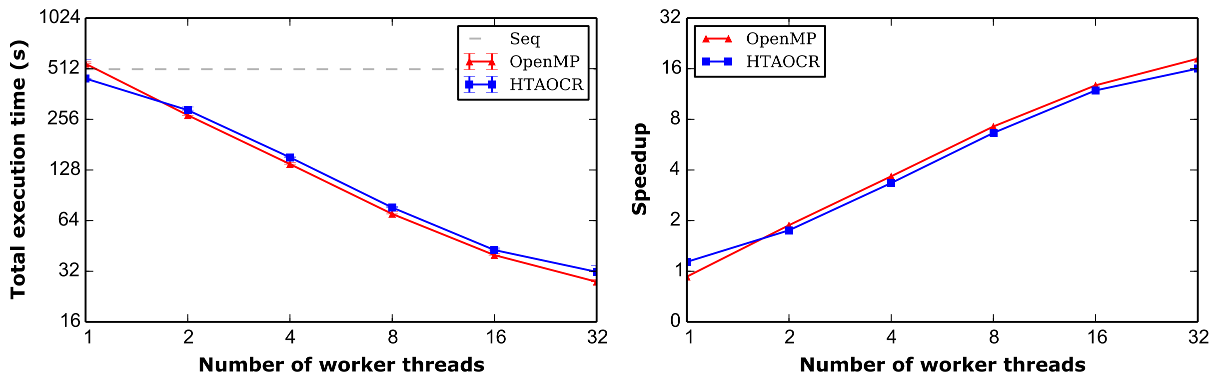


Figure 5.22: CG benchmark result

5.10.6 LU

LU benchmark uses Symmetric Successive Over Relaxation (SSOR) method to solve an equation. It is an iterative method that iterates a fixed number of times to approximate the solution. A sketch of the SSOR algorithm is shown in Algorithm 3. First, the residual is computed in `COMPUTE_RHS`. Next, the lower-triangular system and diagonal systems are formed in `JACLD` and solved in `BLTS`. `JACU` and `BUTS` are for the upper-triangular system. The solution is then updated in `ADD` before the next iteration starts.

The reference OpenMP implementation uses [26] a pipelining method to parallelize `BLTS` and `BUTS`, since the evaluation of a point in the 3-D array depends on the neighboring points and the direction in which the wavefront proceeds. The other operations are parallelized by using parallel for loops straightforwardly. For HTA-OCR, because of the data dependences

on neighboring points, tiles need ghost regions same as mentioned in Section 5.10.4 and ghost region updates are needed when new results are computed. Most of the parallel for loops are converted to `HTA_map()` operations. For `BLTS` and `BUTS`, loops that iterate over HTAs in a wavefront fashion are used, and the tiles on the wavefront are selected to generate subtasks for the computation. For the ghost region updates, each tile is matched with the corresponding neighboring tiles and a subtask is generated with these tiles as input for the updates.

Algorithm 3: SSOR algorithm used in LU benchmark

```
1 for i = 1 to ITMAX do
2     COMPUTE_RHS;
3     JACLD;
4     BLTS;
5     JACU;
6     BUTS;
7     ADD;
8 end
```

Compared to OpenMP, the HTA-OCR version shows good results for lower thread counts, but OpenMP beats it using 32 threads. We think that the better results for lower threads are because of two things: the arrays are tiled in HTA-OCR and may have better performance due to cache effects, and there are chances for the tasks to overlap between iterations, although very limited. At a higher thread count, more tasks are generated while the total work remains the same. The task overhead again becomes relatively more significant and thus the scalability is negatively affected.

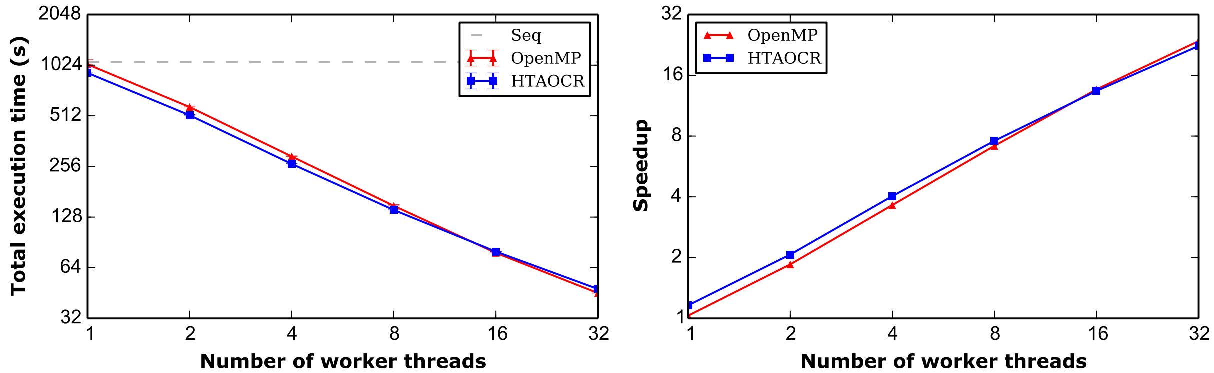


Figure 5.23: LU benchmark result

5.11 Summary

In this chapter, we present various kinds of benchmark programs that we implemented using the HTA-OCR library. For some of them, superior performance results are demonstrated in HTA-OCR due to the benefits provided by the dataflow execution model. There are also some benchmarks ported from OpenMP implementation where we get comparable performance results. For the rest, our current library implementation does not scale well for the fixed problem size. Here, we summarize the important observations from the experimental results.

HTA-OCR performs the best when there are abundant asynchronous subtasks and when the data dependences among the subtasks are sparse. For example, in the tiled Cholesky factorization and the tiled LU factorization benchmarks, we can see greater performance results in HTA-OCR over OpenMP using parallel for loops. All-to-all dependences between operations and dependences on global reduction results can cause the dataflow execution to run in a bulk-synchronous fashion, which is the case in benchmarks such as LULESH and AMGMK Relax. In such cases, the task overlapping in HTA-OCR is limited, and its large overhead of using tasks becomes a disadvantage.

HTA-OCR is much more sensitive to issues related to the task granularity than OpenMP is. To achieve good parallel efficiency, the task granularity has to be large enough to amortize

the overhead, but the overhead is much higher in HTA-OCR. As a result, using the same data decomposition for the same problem in HTA-OCR and in OpenMP can have completely different performance results. For some problems, it is possible to adjust the tiling to reduce the number of tasks and also increase task granularity. However, it is difficult in problems such as LULESH and parallel breadth first search. For LULESH, the task granularity does not only depend on the size of the input tiles but also the type of the operation being performed. Programmers can manually fuse operations to adjust the task granularity, but it is not an ideal solution regarding the programmability. For parallel BFS, decomposing the input data domain does not guarantee balanced load or large enough granularity. A change of the algorithm is needed so that more information can be computed to result in a better dynamic tiling.

In all of our experiments we present the strong scaling results. In fixed-sized problems, the task generation overhead tend to become a problem that limits the performance scalability. When we increase the number of worker threads, we usually partition the HTA into more tiles to expose more parallelism. The amount of overhead in the task generation increases fast when more tiles are used. In the current implementation, the task generation is performed only by the master task, and it easily becomes a bottleneck. In the next chapter, we discuss the possibilities toward a more scalable design.

Chapter 6

Towards A Scalable Design

In the previous chapters, we describe the design and implementation of HTA-OCR in the shared memory environment. We have shown using examples that the programmability of HTA-OCR is similar to that of OpenMP. From the experimental results of various benchmarks, we learn that, for programs with high asynchrony and irregularity, HTA-OCR can deliver greater performance. However, we also identified some issues that limit the scalability of this design. In this chapter, we review the issues and propose possible solutions, either by modifying the HTA-OCR library implementation or adding new features to the OCR runtime. We also discuss potential optimizations before concluding the chapter.

6.1 Overhead Analysis

Before getting to the changes to be made for a more scalable design, we analyze the current design to understand where the overhead comes from. Both the execution time overhead and the storage space overhead are discussed.

Figure 6.1: The effects of task generation overhead

Execution Time Overhead

In the mapping strategy described in Section 3.2, when an HTA program executes, it creates asynchronous subtasks. Hence, there is task generation overhead, which includes the overhead of updating metadata and the overhead of spawning tasks. In the existing implementation, a single master task spawns all the subtasks and carries out all the metadata changes. While it works fine when the thread count and the number of tiles are both low, the task generation is sequential and can become a bottleneck when there is a large amount of subtasks per operation. It can be a problem when the task granularity is either small or large. In Figure 6.1, the top figure illustrates the case when the task granularity is relatively large. We can see that the time spent in the task generation can delay the execution of a group of balanced tasks. It causes an unexpected delay and affects the overall execution time. On the other hand, in the bottom figure, the subtasks have relatively small granularity, resulting in the task generation overhead dominating the total execution time. This limits the strong scalability because when more processing elements are used, we would want to use more tasks, result in increased task generation overhead and decreased task granularity.

We can also see why the sequential task generation is problematic for scalability using an

analytic model to calculate the parallel efficiency. On a machine with P processing elements, consider a simple pointwise addition $C = A + B$ where each of the operand has P tiles (i.e. P tasks are created to utilize P processing elements). Suppose the sequential execution time is T_1, the time it takes to spawn a single task is constant t_g, the time to process the metadata of the operands of a single task is constant t_m, and, as a simplification, there is no other task related overhead. The total execution time is thus:

$$T_p = P \times (t_g + t_m) + \frac{T_1}{P}$$

And the parallel efficiency is:

$$\begin{aligned} E &= \frac{T_1}{P \times T_p} \\ &= \frac{T_1}{P^2 \times (t_g + t_m) + T_1} \end{aligned}$$

With the P^2 term in the denominator, the parallel efficiency decreases quickly when the number of processing elements increases, which explains why it is not scalable. Since it is not possible to completely eliminate the task generation overhead, the best that can be done is to parallelize the task generation so that the quadratic term becomes linear.

One way to parallelize task generation is for OCR to provide a library function for creating a batch of tasks in parallel at the OCR runtime level. This can be useful for data parallel operations that are performed using `HTA_map()`. However, it only allows parallelizing the task spawning but not the metadata processing. Also, in algorithms that use an outer for loop and generate a single task per iteration, such as the Cholesky factorization, the HTA library cannot generate subtasks in a batch. A change in the HTA design is necessary to fully resolve this problem.

Storage Overhead

The HTA programming model represents the tiled arrays as trees. A fixed-sized memory space is allocated for the metadata describing each node in the tree. Given P processing elements, we want to at least partition an array into P leaf tiles so that enough parallelism is exposed. As a result, the storage space needed for metadata is $\Omega(P)$. If the HTA is one-level, the storage requirement is close to the lower bound, and when there are more levels, it can be larger depending on the tiling.

The HTA-MPI implementation runs in SPMD. MPI processes execute the same code and perform HTA operations collaboratively. For every HTA in the program, each process allocates the full tree structure and partial raw data depending on the distribution of the tiles. The raw data does not grow when more processes are used, but the metadata does since it is duplicated everywhere. From the previous analysis, we know that the metadata is $\Omega(P)$ on a single node, and since every process allocates the full tree data structure, the total storage overhead of all processes is $\Omega(P^2)$. This is not scalable if we consider extreme scale machines.

6.2 A Scalable Design

From the overhead analysis, it is clear that the current HTA-OCR design has scalability issue due to both the task generation overhead and the metadata storage overhead. In order to have better scalability, we must parallelize the task generation and reduce the metadata storage requirement.

SPMD + Dataflow Execution

To parallelize the task generation, we propose using SPMD + Dataflow execution, which is similar to the proposal in our previous work [43] but with some modifications. We target a

hypothetical extreme scale computing system which is distributed, hierarchical with powerful computing nodes and each of them has thousands of processing elements that can have shared memory address space within the node. Suppose there are P nodes, we can let P master tasks run in parallel, each executes on one of the nodes and the same HTA program. A distribution function is needed to map the disjoint subsets of the data tiles in an HTA to the master tasks. While the distribution function of an HTA in the shared memory implementation has no effect on the execution, in the distributed memory environment, the distribution function and the tiling together express the data locality the programmer wants. When a set of tiles are mapped to a master task, it can mean that the tasks operating on the set of tiles have data locality and should stay close if possible. The HTA library implementation could supply affinity hints to the OCR runtime while spawning the tasks to achieve locality optimization. Figure 6.2 shows an example of SPMD+Dataflow execution when $P = 3$.

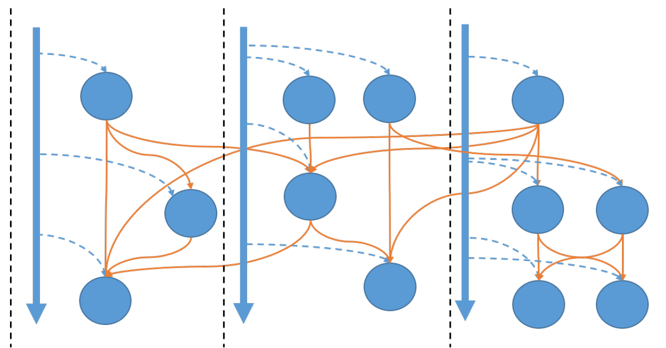


Figure 6.2: SPMD + Dataflow execution with multiple master tasks

The master tasks execute the HTA program sequentially until they encounter HTA operations. Instead of generating subtasks for all tiles in the LHS as in the shared memory design, they only spawn subtasks for the tiles they own (i.e. local tiles) on the LHS, following the *owner computes* rule[1]. The master tasks do not wait for the completion of the local asynchronous subtasks unless one of their statements depends on the subtask results. This way, the task generation is parallelized. The execution model is different from SPMD

[1]A tile is owned by a master task when it is mapped to the master task by the distribution function.

since the asynchronous subtasks can synchronize with each other without joining back to the master tasks.

Reducing Metadata Overhead

For the metadata storage overhead, a scalable design that runs in SPMD fashion should not have to store a complete copy of the HTA metadata but only store the metadata of its local tiles and ancestor nodes. If we limit HTA shapes to be regular, we do not have to store the full HTA tree, because we can design a fast distribution function to locate the tiles quickly without traversing the tree structure. Given the tile indices, a good distribution function should be able to determine whether a tile is local or remote in $O(1)$. Complex distribution functions for HTAs of irregular shapes can still be supported, but the users should be warned that the use of them could make the application less scalable.

Task Synchronization

In the existing implementation, there is only one master task, and it is the point of serialization for all metadata changes of a tile. The requests (i.e. dependences) for a tile are processed by the master task in program order, so the master task can ensure the subtasks requesting the same tile executes in the order of the corresponding request occurrences. However, when there are multiple master tasks, no single point of serialization exists anymore. A new strategy is needed to make sure that the master tasks can specify the correct events to express the data dependences of the subtasks without frequently synchronizing with each other. For this purpose, we can rely on a set of OCR API functions to create OCR events with user specified GUIDs. Essentially, it allows us to create uniquely named events (each event represents a dependence) in a distributed environment. With this mechanism, we can have a protocol of naming the events representing for the tile data dependences in the HTA program execution, so that by following the protocol, the master tasks can generate the

names of events without synchronizing with each other.

As an example for such protocol, we can use a hash function HASH(operation_count,task_ord,operand_pos) to generate a GUID that uniquely represents a data dependence edge in the task graph, and rely on OCR's global addressing ability to synchronize in a distributed environment. operation_count is the number of HTA operations that have been executed. task_ord is the ordinality of the task in the operation. operand_pos represents the type and ordinality of the operand of the corresponding task. Unique names can be generated from these values to identify data dependences of a task.

Argument	Set of Values
operation_count	{0, 1, 2, ...}
task_ord	{0, 1, 2, ...}
operand_pos	{INPUT0, INPUT1, ..., OUTPUT0, OUTPUT1, ...}

Table 6.1: The values of the arguments to the hash function to generate unique GUID

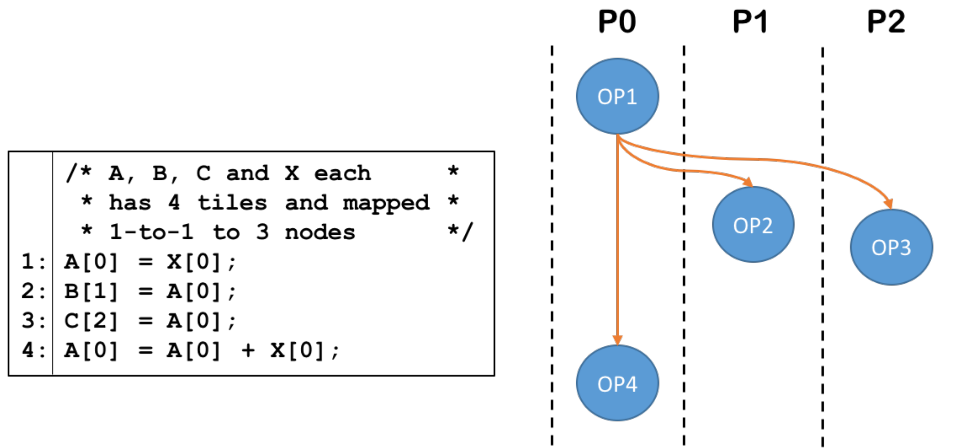
When a master task finds a new HTA operation, it determines whether the tiles accessed by the operation are local or remote by evaluating the distribution function. If a local tile is on the LHS, it creates an asynchronous subtask to perform the operation. If a local tile is on the RHS, the master task may need to supply this tile to a remote task. Lets consider an example shown in Figure 6.3. In Figure 6.3a, the example code is given, and the ideal task graph that captures the data dependence in the program is also shown. P<number> represents the nodes and also the master tasks that execute on them, and OP<number> represents the subtasks that performs the operations.

Now, if we consider the implementation details, it gets trickier than the seemingly simple ideal task graph. In practice, how does the task OP1 synchronizes with OP2 and OP3? When the master task on P0 executes the program and sees the first operation, it creates OP1 asynchronously. But at the creation time of OP1, P0 is not aware of the future tasks that depends on the result of OP1. As a result, OP1 cannot directly satisfy the dependence slots of OP2 and OP3 when it completes. There are two possible solutions, depending on whether

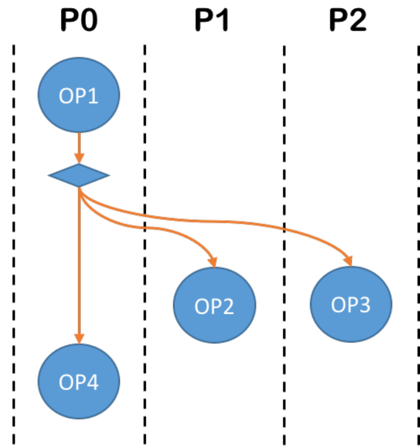
all the master tasks maintain metadata of all the tiles.

When the master tasks maintain the metadata of all the tiles, the solution is simpler. When `P0` creates `OP1`, it creates a named sticky event as the placeholder of the output from `OP1`. Since other master tasks also maintain the metadata of `A[0]`, when they execute the first operation, they can use the hash function call `HASH(1,0,OUTPUT0)` to compute the GUID of the output event and stores the GUID in the metadata. Later, when they need to create a new task that depends on the latest `A[0]` content, they simply read the local `A[0]` tile metadata and know which event represents for the desired data. Figure 6.3b illustrates this scenario. However, as we have pointed out previously, only when the HTA metadata is fully distributed can we minimize the storage overhead. Therefore, this solution is not suitable for extreme scale systems.

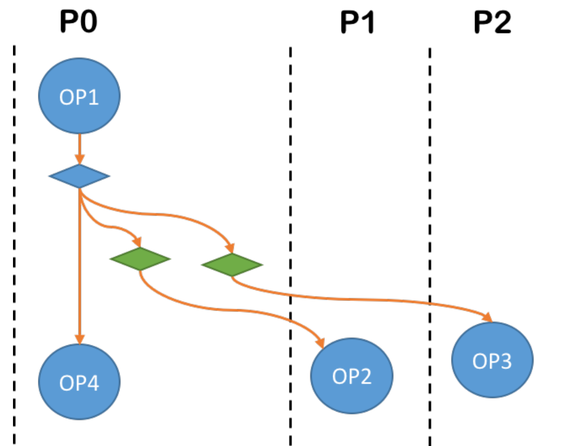
If the metadata is fully distributed, the master tasks only possess the metadata of local tiles. This means that `P1` and `P2` do not know the named event representing the completion of the last user of `A[0]` which is the task `OP1`. They can only use the hash function to compute unique names of the data dependences of their local tasks. For example, `P1` calls `HASH(2,0,INPUT0)` and `P2` calls `HASH(3,0,INPUT0)`. On the other hand, `P0` notices that it has to supply data when it executes the second and the third operation, and it calls `HASH(2,0,INPUT0)` and `HASH(3,0,INPUT0)` correspondingly to get the same GUIDs `P1` and `P2` generate without synchronizations. At the time these data dependences are discovered by `P0`, the desired output may not be ready yet. `P0` creates special *proxy* events, named with the GUIDs generated by the hash function calls, as placeholders for the data produced in `OP1` and to pass the data along to the remote subtasks, as shown in Figure 6.3c. Notice that the proxy events are owned by the supplier `P0`, because it is the owner of `A[0]` and it knows how the to connect the proxy events to the output event of `OP1` represented by `HASH(1,0,OUTPUT0)`, where the desired `A[0]` copy is held. In this way, we avoid having every master task maintaining full HTA metadata.



(a) Example code and the ideal task graph



(b) Task synchronization of the example code with OCR events when every master task maintains full HTA metadata



(c) Task synchronization of the example code if master tasks only maintain local tile metadata

Figure 6.3: Illustrations of the task synchronization in SPMD+Dataflow for the given program

6.3 Performance Optimizations

In the previous section, the fundamental mechanisms of building a scalable HTA-OCR implementation have been described. Here, we further discuss the potential optimizations that can be added to the implementation, either in the HTA library code or the OCR runtime. We also discuss possible compiler optimizations.

6.3.1 Support for Parallel Reductions

Parallel reduction is a very common pattern that appears in parallel algorithms. Currently, HTA provides two types of reductions, one to reduce a whole HTA into a scalar value, and the other to reduce an HTA in a specific dimension. The HTA library creates OCR tasks to perform reductions on partial data, and then, depending on the amount of tasks, a sequential reduction or a tree reduction of the partial results is used.

Implementing the reduction operation at the HTA library level using general OCR tasks does not grant optimal performance, because the amount of tasks and events that are needed for very small amount of computation can be quite large. On the other hand, there are often machine-dependent ways to perform optimal parallel reduction on different classes of machines. Some high-performance computing systems even come with specialized interconnections [20] in the hardware just for reductions. It is better that OCR defines an API function for reduction to a scalar and the HTA library builds upon that, so that the OCR runtime implementations can provide optimized support for reductions.

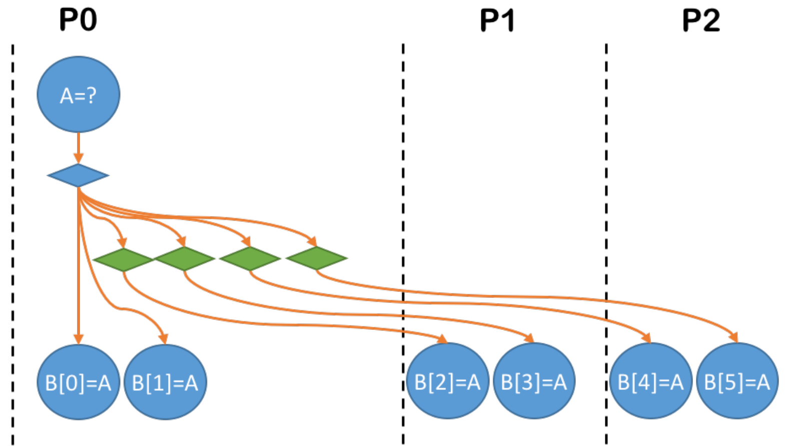
Besides reducing to a single scalar, reduction of a fixed-sized array is also very common. For example, in the tiled matrix-matrix multiplication algorithm, the tasks that writes to the same tile forms a dependence chain. Here, the tasks in a chain accumulates to the same tile and it is essentially a reduction of tiles. If OCR provides a built-in function to create a subgraph of reduction tasks, the HTA library can exploit it. The same argument can be

made for other collective operations, such as prefix scan, all-to-all exchange, ...etc.

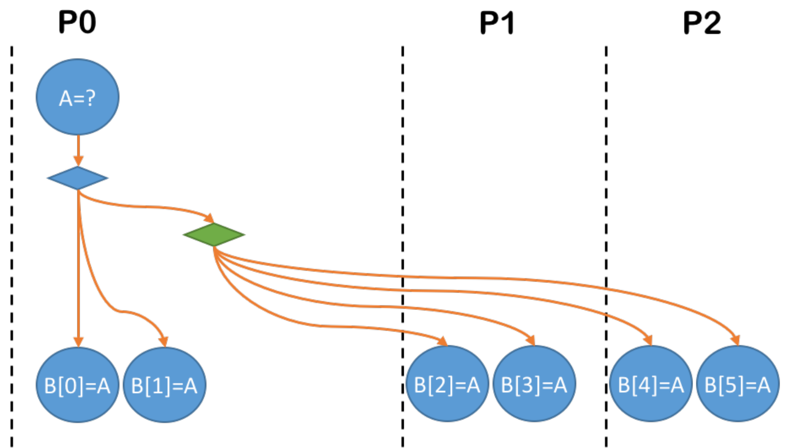
6.3.2 Curtailing Proxy Events

The task synchronization mechanism described in Section 6.2 introduces the proxy events, which are necessary for the data dependences across master tasks if they only maintain local tile metadata. The number of proxy events in an operation can be very large. But when multiple subtasks of the same operation depend on the same tile, the number of proxy events can be reduced. For example, lets consider an HTA `A` which is a single tile, and an HTA `B` which has six tiles of the same size as `A`, and the tiles are mapped in block distribution on to three nodes. An assignment operation `B = A` broadcasts the content of `A` to overwrite all the tiles in `B`. As shown in Figure 6.4, if we use the basic design and create a proxy event for each data dependence, four proxy events are needed to pass the data to remote tasks. However, since the individual tile assignments are expressed in one assignment operation, the library can dynamically detect, within the scope of a single HTA operation, that four remote subtasks have the same input dependence and thus create only one proxy event in `P0` to pass the requested tile to them. This optimization can be implemented in the HTA library.

If the assignment operation is manually unrolled, the library optimization will not work, since the library processes a single HTA operation at a time, and it cannot see beyond the scope of one operation. For instance, it is possible that the program performs assignments of `A` to the odd tiles `B[1] = A; B[3] = A; B[5] = A;`. The library would still create two proxy events for the remote tasks that assign `B[3]` and `B[5]`. In cases like this, a source-to-source compiler may merge the individual tile assignments into a form that the library can still optimize.



(a) Task graph with basic design. Four proxy events are used.



(b) Task graph when a single event is used for the subtasks of the same operation depending on the same tile.

Figure 6.4: Optimization to curtail the number of proxy events at the HTA library level

6.3.3 Graph Reduction

As we have demonstrated in the experiments, the task granularity can have an enormous impact on the performance results. If the programmer attempts over-decomposition to expose more parallelism in the program, the added overhead may negatively affect the performance. For some algorithms, it might be easy to determine a good tiling and strike a balance between task granularity and parallelism, such as the numerical algorithms dealing with dense data. But for others, this might be difficult due to the nature of data dependent workloads.

It may take a lot of time to fine-tune an algorithm for specific machines or dataset. If an HTA compiler exists, programmers may express the finest-grain parallelism using HTAs, and let the HTA compiler analyze the dependence graph and optimize the graph by reducing the graph composed of fine-grain tasks into optimal grain size for the underlying machines and runtime system. This is an interesting and complicated problem which beyond the scope of this work.

Chapter 7

Related Work

In this chapter, we discuss the related work, including the runtime systems that are based on dataflow-like principles for their execution models, and the programming models that are built upon dataflow runtime systems.

7.1 SWARM

SWift Adaptive Runtime Machine (SWARM) [30] is a runtime system that uses the codelet program execution model in its design. The project aims to design a highly sophisticated runtime system that can intelligently make scheduling decisions for future exascale computing systems with heterogeneous processors, so that programmers only have to focus on exposing as much parallelism in the system as possible. A SWARM program is formed by using fine-grained unit of work called codelets to construct a codelet dependence graph. The concepts are very similar to those used in OCR design: codelets are analogous to OCR event-driven tasks, and the codelet dependence graphs are analogous to the task graphs in OCR. The programming interface of SWARM is quite different from that of OCR. Users can either use the SWARM API, which is a low-level language that lets user explicitly define codelets

including their computations and their interactions, or use a higher-level language extension to C, the SWARM Codelet Association LanguagE (SCALE). In a software stack, SWARM is at the same level as the OCR runtime system. SWARM's distributed system support is different from OCR since it does not have support for global addressable objects. Explicit message passing is needed. We evaluated using SWARM as the HTA library backend but ended up with OCR. We use the SWARM trace viewer to visualize the execution of our HTA-OCR applications.

7.2 Charm++

Charm++ [27] is a parallel programming paradigm which deploys a message driven execution model. Chares are C++ objects that can contain methods and data, and the programmer builds an application by defining chares and explicitly program their interactions using method invocations (i.e. messages). There can be multiple chare objects mapped to a processor, and a Charm++ scheduler on the processor selects the method invocations to be scheduled on the cores for execution. An adaptive runtime system is responsible for mapping chare objects to processors, and it can perform various runtime optimizations such as object migration for data locality and load balancing. Compared to OCR and SWARM, Charm++ has a higher-level programming interface that uses the object-oriented paradigm. The idea of the message driven execution is very similar that of the dataflow execution model, except that the method invocations can also represent for control dependences. Compared with the HTA programming model, the chares allow more flexibility in expressing various kinds of computations, not limited to array operations. However, since the method invocations are explicitly programmed, the Charm++ language has a lower level programming interface and the application code has the resemblance of a dependence graph instead of a structured imperative program.

7.3 Legion

Legion programming system [6] includes a runtime system and API. To write programs for Legion, the programmer decomposes application data into *logical regions* and spawns asynchronous tasks that operate on the regions. The regions can be partitioned and represented in a hierarchical way as needed. The tasks are scheduled for execution when their data dependences on logical regions are satisfied, so we also categorize the execution model as dataflow. One important difference of Legion from the other runtime systems we mentioned is that the dependences between tasks are not explicitly programmed. In OCR, explicit dependence satisfactions are inserted by programmers. In Legion, the programmer spawns tasks with input and output logical regions, and the Legion runtime has a software out-of-order processor mechanism that dynamically determines the data dependences between tasks due to their region usage and the order of the task executions. Legion is at a lower level than HTA, since in HTA, parallel tasks are implicitly created when HTA operations are performed. Regent [40] is a higher-level programming language built upon Legion.

7.4 OpenMP Tasking with Data Dependent Tasks

In the OpenMP 3.0 specification, OpenMP Tasking was introduced to provide ways to program unstructured parallelism. It enabled task parallelism but was not flexible enough to express maximal parallelism in the program because it still relied on barriers. Based on the encouraging results of SPMSs [35], the extension of OpenMP Tasking to include support for data dependent tasks is proposed in [16] by introducing new clauses for the `#pragma omp task` construct, including using the clauses `input`, `output` and `inout` in combination with data reference lists. With these hints, the dependence relationships between tasks can be derived at runtime, and barriers are not always necessary. As a result, there are more chances for task overlapping at runtime. This is exactly what the dataflow execution

model does. The proposal was eventually accepted and incorporated with some modifications into the OpenMP 4.0 specification. Compared with HTA-OCR, the data dependent tasks in OpenMP has a limited scope within a parallel section. Thus, it may not be suitable for large applications but only for smaller code segments that needs to be accelerated. In Section 5.4, we compared our HTA-OCR Sparse LU factorization implementation with an OpenMP Tasking implementation and found the performance results comparable.

7.5 Tensorflow

Tensorflow [1] is a programming interface for machine learning algorithms. It uses an extension of the dataflow graph called stateful dataflow graph to express computations. In such a graph, the nodes are connected by directed edges, and the values that flow through the edges are tensors. It allows the existence of persistent mutable tensors in the system in some special operations. And it also defines special edges that represents the control dependences so that the programmer can enforce a happen-before relationship between nodes. It has implementations for both single node and multiple node executions. Users can provide kernels, which can be machine-dependent implementations of the same operation for heterogeneous devices, so that users can adapt their programs for different systems with little modifications. Compared with the other runtime systems we have mentioned, Tensorflow does not construct the dataflow graph with the progress of execution. Instead, it requires the graph to be constructed before the computation starts. The benefit of this approach is that since the graph is constructed before computation, the runtime system can optimize the graph structure and has more information to make better arrangements on how the nodes should be scheduled. But it will not work well for general purpose programming since the graph can be indefinitely large. Tensorflow also requires explicit parallelization of computations. Although the dataflow graph representation allows the runtime to make intelligent scheduling

decisions of the nodes, to exploit the finer-grain parallelism in the program, the programmer needs to explicitly decompose subgraph containing coarse-grain operations into subgraphs containing finer-grain partial computations.

7.6 PaRSEC

Parallel Runtime Scheduling and Execution Controller (PaRSEC) [10] is a runtime system that schedules computation tasks driven by data dependences. It is designed targeting manycore, distributed and heterogeneous large-scale machines. The runtime system executes a DAG represented as a low-level textual Job Data Flow (JDF) format. The user does not have to explicitly program in JDF. Instead, they can write sequential program or using higher-level code such as StarSS [36] and use a compiler to compile the code into JDF for the PaRSEC runtime system to execute. In contrast to OCR which requires the user to explicit insert function calls to create tasks, PaRSEC creates tasks automatically as dictated by data dependences, resulting in lower memory overhead. It can be interesting to explore the possibility of adapting HTA as a high-level abstraction of PaRSEC.

Chapter 8

Future Work

There are many interesting possible extensions to the work presented in this thesis. Here, we discuss a few possibilities.

Distributed Implementation

The most important future work for HTA-OCR is to extend the support to distributed memory machines. If we want the HTA programming model to be practical, it has to support large scale machines which are mostly distributed memory systems. While OCR does provide support on distributed memory system, our current implementation only supports running on single node shared memory machine because the split-phase continuation mechanism implemented in the HTA-OCR library does not work in a distributed memory environment. A native OCR runtime support for continuations on distributed systems is needed.

There can be many new issues rising from the attempt to move to a distributed memory environment which need the library developers to work closely together with the runtime system developers to solve. For example, in Chapter 6, we proposed a distributed design that relies heavily on the OCR events being globally addressable. But at the runtime system level, how are the globally addressable objects migrate from a node to another and how much

overhead does it incur? How much overhead does a remote event satisfaction incur? When is a data block propagated to a remote node? What happens when the tasks on the same node request the same remote data block? There are lots of opportunities for optimizations both at the library level and at the runtime level waiting to be explored.

Support for Other Dataflow Runtime Systems

Another interesting topic is whether our design is general enough for other dataflow-like runtime systems. We mentioned many runtime systems that adopt the ideas of the dataflow execution model, but they all have significant differences in their runtime programming interface design. For example, Charm++ uses an object-oriented programming interface and the dependences are represented as method invocations on chares. This is significantly different from OCR. It is interesting to investigate whether we can generalize programming interface of different runtime systems and present a middle layer for the HTA library to build on top of them. The Parallel Intermediate Language (PIL) [41] attempts to present an unified programming interface for not just the dataflow runtime systems but also for other parallel programming models. HTA was implemented on PIL, but the implementation was limited by the parallel program construct provided by PIL and does not express maximal parallelism as HTA-OCR does.

HTA Compiler

As mentioned in 6.3, there are some optimizations which cannot be done at the library level. An HTA compiler can analyze the access pattern on the HTA tiles and perform complex optimizations. This can be done either with a source-to-source HTA compiler that takes the user HTA program and generates an optimized HTA program, or a source-to-source HTA compiler that takes the user HTA program and generates an OCR program. Various kinds of optimizations can be useful. For example, code motion can rearrange the split-

phase continuation so that more asynchronous tasks are generated before the split-phase action. A compiler can also detect reduction pattern like the one in the tiled matrix-matrix multiplication code in Section 5.2 and replace it with the library reduction routine.

Chapter 9

Conclusions

The dissertation presents an adaptation of the HTA programming model upon the dataflow execution model. 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. For performance evaluation, a variety of benchmark programs were implemented using the HTA-OCR API and the experiments were conducted. With the tracing API, program execution traces can be generated without interfering with the timing of the actual task execution, so that the execution behavior can be observed easily. The results were analyzed and precious lessons were learned about the types of the execution patterns that can benefit from dataflow executions and the types that prevent us from gaining good performance results. We also discovered some issues in our design that potentially limit the scalability, and proposed solutions for them.

With the HTA-OCR API, programmers now have a higher-level way of building applications on top of OCR. The productivity can be significantly improved because programmers no longer have to decompose computations into task dependence graphs and write their

programs by explicitly constructing the graphs using the OCR API. Instead, they can create HTAs and express their algorithms in the imperative programming style using the array notations and rely on the HTA-OCR library to convert their HTA programs to OCR task graphs dynamically. From the example programs in the appendix, we see that the lines of code in the HTA-OCR version can be less than 50% of the OCR counterpart. This can be further improved if C++ was used.

Our work provides insights into library development upon the dataflow execution model. The insights are not only useful for the library developers but also for the runtime system developers. For the library developers, the analysis of the overhead in our own design can be something that they take into consideration before actually starting their implementations. For the runtime system developers, they can learn what runtime API features can facilitate the library or application development. Our discussions about the issues that hinder the scalability can be generalized and considered for any software built upon dataflow runtime systems.

Appendix A

Tiled Matrix-matrix Multiplication Full Programs

Listing A.1: Pure OCR tiled matrix-matrix multiplication implementation

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <math.h>
4 #include "ocr.h"
5
6 // pointer to data (NOTICE: this won't work on distributed system)
7 ocrGuid_t** A;
8 ocrGuid_t** B;
9 ocrGuid_t** C;
10 double* A_flat;
11 double* B_flat;
12 double* C_flat;
13 double*** A_tiles;
14 double*** B_tiles;
15 double*** C_tiles;
16
17 ocrGuid_t
18 localMMEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[])
19 {
20   double *a, *b, *c;
21   int n = paramv[0];
22   c = (double*)depv[0].ptr;
23   a = (double*)depv[1].ptr;
24   b = (double*)depv[2].ptr;
25
26   for (int i = 0; i < n; i++) // sequential local matrix multiplication
27     for (int j = 0; j < n; j++)
28       for (int k = 0; k < n; k++)
29         c[i * n + j] += a[i * n + k] * b[k * n + j];
30   return NULL_GUID;
31 }
32
33 ocrGuid_t
34 mainLoopTask(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[])
```

```
35 {
36   int nBlocks = paramv[0];
37   int nEntries = paramv[1];
38   long int matrixWidth = nBlocks * nEntries;
39   for (int i = 0; i < nBlocks; i++) {
40     for (int j = 0; j < nBlocks; j++) {
41       double* ptr_tile = NULL;
42       ptr_tile = A_tiles[i][j];
43       for (int x = 0; x < nEntries; x++)
44         for (int y = 0; y < nEntries; y++)
45           ptr_tile[x * nEntries + y] =
46             A_flat[(i * nEntries + x) * matrixWidth + (j * nEntries + y)];
47       ptr_tile = B_tiles[i][j];
48       for (int x = 0; x < nEntries; x++)
49         for (int y = 0; y < nEntries; y++)
50           ptr_tile[x * nEntries + y] =
51             B_flat[(i * nEntries + x) * matrixWidth + (j * nEntries + y)];
52       ptr_tile = C_tiles[i][j];
53       for (int x = 0; x < nEntries; x++)
54         for (int y = 0; y < nEntries; y++)
55           ptr_tile[x * nEntries + y] =
56             C_flat[(i * nEntries + x) * matrixWidth + (j * nEntries + y)];
57     }
58   }
59   ocrGuid_t** A = (ocrGuid_t**)paramv[2];
60   ocrGuid_t** B = (ocrGuid_t**)paramv[3];
61   ocrGuid_t** C = (ocrGuid_t**)paramv[4];
62
63   ocrGuid_t completion_events[nBlocks][nBlocks][nBlocks];
64
65   ocrGuid_t templateLocalMM;
66   ocrGuid_t firstRoundSubTasks[nBlocks][nBlocks];
67   ocrGuid_t subtaskguid;
68
69   ocrEdtTemplateCreate(&templateLocalMM, localMMEdt, 1, 4);
70   ocrHint_t priority_hint;
71   ocrHintInit(&priority_hint, OCR_HINT_EDT_T);
72
73   // Build DFG of Matrix multiplication
74   for (int k = 0; k < nBlocks; k++) {
75     for (int i = 0; i < nBlocks; i++) {
76       for (int j = 0; j < nBlocks; j++) {
77         int paramc = 1;
78         int depc = 4;
79         u64 paramv[1];
80         ocrGuid_t depv[4];
81         paramv[0] = nEntries; // block width
82         depv[0] = C[i][j];
83         depv[1] = A[i][k];
84         depv[2] = B[k][j];
85         depv[3] =
86           (k == 0)
87             ? UNINITIALIZED_GUID
88             : completion_events[i][j][k - 1]; // hold the firing of k==0 tasks
89         ocrSetHintValue(&priority_hint, OCR_HINT_EDT_PRIORITY, nBlocks - k);
90         ocrEdtCreate(&subtaskguid, templateLocalMM, EDT_PARAM_DEF, paramv,
91                      EDT_PARAM_DEF, depv, EDT_PROP_NONE, &priority_hint,
92                      &completion_events[i][j][k]);
93         if (k == 0)
94           firstRoundSubTasks[i][j] = subtaskguid;
95       }
96     }
97   }
98
99   // Fire the first round
100  for (int i = 0; i < nBlocks; i++) {
101    for (int j = 0; j < nBlocks; j++) {
102      ocrAddDependence(NULL_GUID, firstRoundSubTasks[i][j], 3, DB_MODE_RW);
103    }
104  }
105
```

```
106   ocrEdtTemplateDestroy(templateLocalMM);
107   return NULL_GUID;
108 }
109
110 ocrGuid_t
111 wrapUpTask(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[])
112 {
113   ocrShutdown();
114   return NULL_GUID;
115 }
116
117 ocrGuid_t
118 mainEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[])
119 {
120   int argc = getArgc(depv[0].ptr);
121   char** argv = (char**)malloc(argc * sizeof(char*));
122   for (int i = 0; i < argc; i++) {
123     char* arg = getArgv(depv[0].ptr, i);
124     argv[i] = arg;
125   }
126
127   int nBlocks = atoi(argv[1]);
128   int nEntries = atoi(argv[2]);
129   long int matrixWidth = nBlocks * nEntries;
130   long int matrixSize = matrixWidth * matrixWidth;
131
132   // Allocate un-tiled A, B and C matrices
133   A_flat = (double*)malloc(sizeof(double) * matrixSize);
134   B_flat = (double*)malloc(sizeof(double) * matrixSize);
135   C_flat = (double*)malloc(sizeof(double) * matrixSize);
136
137   srand(time(NULL));
138   // Initialize data
139   for (int i = 0; i < matrixWidth; i++) {
140     for (int j = 0; j < matrixWidth; j++) {
141       A_flat[i * matrixWidth + j] = (rand() % 100) / 100.0;
142       B_flat[i * matrixWidth + j] = (rand() % 10) / 10.0;
143       C_flat[i * matrixWidth + j] = 0.0;
144     }
145   }
146
147   // Allocate tiled matrices and initialize data
148   ocrGuid_t tmp_guid;
149   ocrDbCreate(&tmp_guid, (void**)&A, sizeof(ocrGuid_t*) * nBlocks, DB_PROP_NONE,
150               NULL_HINT, NO_ALLOC);
151   ocrDbCreate(&tmp_guid, (void**)&B, sizeof(ocrGuid_t*) * nBlocks, DB_PROP_NONE,
152               NULL_HINT, NO_ALLOC);
153   ocrDbCreate(&tmp_guid, (void**)&C, sizeof(ocrGuid_t*) * nBlocks, DB_PROP_NONE,
154               NULL_HINT, NO_ALLOC);
155   A_tiles = (double***)malloc(sizeof(double**) * nBlocks);
156   B_tiles = (double***)malloc(sizeof(double**) * nBlocks);
157   C_tiles = (double***)malloc(sizeof(double**) * nBlocks);
158   for (int i = 0; i < nBlocks; i++) {
159     ocrDbCreate(&tmp_guid, (void**)&A[i], sizeof(ocrGuid_t) * nBlocks,
160                 DB_PROP_NONE, NULL_HINT, NO_ALLOC);
161     ocrDbCreate(&tmp_guid, (void**)&B[i], sizeof(ocrGuid_t) * nBlocks,
162                 DB_PROP_NONE, NULL_HINT, NO_ALLOC);
163     ocrDbCreate(&tmp_guid, (void**)&C[i], sizeof(ocrGuid_t) * nBlocks,
164                 DB_PROP_NONE, NULL_HINT, NO_ALLOC);
165     A_tiles[i] = (double**)malloc(sizeof(double*) * nBlocks);
166     B_tiles[i] = (double**)malloc(sizeof(double*) * nBlocks);
167     C_tiles[i] = (double**)malloc(sizeof(double*) * nBlocks);
168     for (int j = 0; j < nBlocks; j++) {
169       double* ptr_tile = NULL;
170       ocrDbCreate(&A[i][j], (void**)&ptr_tile,
171                   sizeof(double) * nEntries * nEntries, DB_PROP_NONE, NULL_HINT,
172                   NO_ALLOC);
173       A_tiles[i][j] = ptr_tile;
174
175       ocrDbCreate(&B[i][j], (void**)&ptr_tile,
176                   sizeof(double) * nEntries * nEntries, DB_PROP_NONE, NULL_HINT,
```

```
177                 NO_ALLOC);
178       B_tiles[i][j] = ptr_tile;
179
180       ocrDbCreate(&C[i][j], (void**)&ptr_tile,
181                   sizeof(double) * nEntries * nEntries, DB_PROP_NONE, NULL_HINT,
182                   NO_ALLOC);
183       C_tiles[i][j] = ptr_tile;
184     }
185   }
186   // Create a finish EDT to run the 3 level loop
187   ocrGuid_t templateMainLoop;
188   ocrGuid_t mainLoopTask_guid;
189   ocrGuid_t mainLoopFinish;
190   u32 new_paramc = 5;
191   u64 new_paramv[new_paramc];
192   new_paramv[0] = nBlocks;
193   new_paramv[1] = nEntries;
194   new_paramv[2] = (u64)A;
195   new_paramv[3] = (u64)B;
196   new_paramv[4] = (u64)C;
197
198   u32 new_depc = 1;
199   ocrGuid_t new_depv[new_depc];
200   new_depv[0] = UNINITIALIZED_GUID;
201
202   ocrEdtTemplateCreate(&templateMainLoop, mainLoopTask, new_paramc, new_depc);
203
204   ocrEdtCreate(&mainLoopTask_guid, templateMainLoop, EDT_PARAM_DEF, new_paramv,
205                EDT_PARAM_DEF, new_depv, EDT_PROP_FINISH, NULL_HINT,
206                &mainLoopFinish);
207
208   // Create a wrap up task that depends on mainLoopFinish event, which
209   // calls ocrShutdown
210   ocrGuid_t templateWrapUp;
211   ocrGuid_t wrapUp;
212   new_paramc = 2;
213   new_paramv[0] = nBlocks;
214   new_paramv[1] = nEntries;
215   new_depc = 1;
216   new_depv[0] = mainLoopFinish;
217
218   ocrEdtTemplateCreate(&templateWrapUp, wrapUpTask, new_paramc, new_depc);
219   ocrEdtCreate(&wrapUp, templateWrapUp, EDT_PARAM_DEF, new_paramv,
220                EDT_PARAM_DEF, new_depv, EDT_PROP_NONE, NULL_HINT, NULL);
221
222   ocrAddDependence(NULL_GUID, mainLoopTask_guid, 0, DB_MODE_RW);
223
224   return NULL_GUID;
225 }
```

Listing A.2: HTA-OCR tiled matrix-matrix multiplication implementation

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <time.h>
 4 #include <string.h>
 5 #include <math.h>
 6 #include "HTA.h"
 7 #include "HTA_timer.h"
 8 #include "HTA_operations.h"
 9 #include "Tuple.h"
10 #include "Distribution.h"
11 #include "test.h"
12
13 void
14 MATMUL(HTA** lhs, HTA** rhs, uint64_t* capture)
15 {
```

```
16   double* a = (double*)HTA_get_ptr_raw_data(rhs[0]);
17   double* b = (double*)HTA_get_ptr_raw_data(rhs[1]);
18   double* c = (double*)HTA_get_ptr_raw_data(lhs[0]);
19   int m = lhs[0]->flat_size.values[0]; // assume square tile
20
21   for (int i = 0; i < m; i++)
22     for (int j = 0; j < m; j++)
23       for (int k = 0; k < m; k++)
24         c[i * m + j] += a[i * m + k] * b[k * m + j];
25 }
26 int
27 hta_main(int argc, char** argv)
28 {
29   int TILES = atoi(argv[1]);
30   int nEntries = atoi(argv[2]);
31   long int MATRIX_WIDTH = TILES * nEntries;
32   long int MATRIX_SIZE = MATRIX_WIDTH * MATRIX_WIDTH;
33   double* A = (double*)malloc(sizeof(double) * MATRIX_SIZE);
34   double* B = (double*)malloc(sizeof(double) * MATRIX_SIZE);
35   double* R1 = (double*)malloc(sizeof(double) * MATRIX_SIZE);
36   double* R2 = (double*)malloc(sizeof(double) * MATRIX_SIZE);
37   int i, j, k, err;
38
39   int pid = MYRANK;
40
41   Tuple mesh = HTA_get_vp_mesh(2);
42
43   Dist dist;
44   Dist_init(&dist, DIST_BLOCK, &mesh);
45   Tuple flat_size = Tuple_create(2, MATRIX_WIDTH, MATRIX_WIDTH);
46
47   // create an empty shell
48   HTA* h1 =
49     HTA_create_with_pid(pid, 2, 2, &flat_size, 0, &dist, HTA_SCALAR_TYPE_DOUBLE,
50                         1, Tuple_create(2, TILES, TILES));
51   HTA* h2 =
52     HTA_create_with_pid(pid, 2, 2, &flat_size, 0, &dist, HTA_SCALAR_TYPE_DOUBLE,
53                         1, Tuple_create(2, TILES, TILES));
54   HTA* result =
55     HTA_create_with_pid(pid, 2, 2, &flat_size, 0, &dist, HTA_SCALAR_TYPE_DOUBLE,
56                         1, Tuple_create(2, TILES, TILES));
57
58   srand(time(NULL));
59   // create a 2D matrix
60   for (i = 0; i < MATRIX_WIDTH; i++)
61     for (j = 0; j < MATRIX_WIDTH; j++) {
62       // A[i*MATRIX_WIDTH + j] = (j==i)? 1:0;
63       A[i * MATRIX_WIDTH + j] = (rand() % 100) / 100.0;
64       B[i * MATRIX_WIDTH + j] = (rand() % 10) / 10.0;
65       R1[i * MATRIX_WIDTH + j] = 0;
66       R2[i * MATRIX_WIDTH + j] = 0;
67     }
68
69   // initialize the HTA using 2D matrix
70   HTA_init_with_array(h1, A);
71   HTA_init_with_array(h2, B);
72   HTA_init_with_array(result, R1);
73
74   for (k = 0; k < TILES; k++) {
75     for (i = 0; i < TILES; i++) {
76       for (j = 0; j < TILES; j++) {
77         Tuple ij = Tuple_create(2, i, j);
78         Tuple ik = Tuple_create(2, i, k);
79         Tuple kj = Tuple_create(2, k, j);
80         HTAOCR_SET_PRIORITY(HTAOCR_EDT_DEFAULT_PRIORITY - k);
81         HTA_map(MATMUL, ARGS(LHS(0, HTA_pick_one_tile(result, &ij)),
82                              RHS(0, HTA_pick_one_tile(h1, &ik),
83                                     HTA_pick_one_tile(h2, &kj))));
84       }
85     }
86   }
```

```
87
88   HTA_UNCLAIM_RET(1, result);
89   HTA_FLATTEN_RET(R1, NULL, NULL, result);
90
91   free(A);
92   free(B);
93   free(R1);
94   free(R2);
95   HTA_destroy(h1);
96   HTA_destroy(h2);
97   HTA_destroy(result);
98   return 0;
99 }
```

Listing A.3: Pure OpenMP tiled matrix-matrix multiplication implementation

```
1  #include <stdio.h>
2  #include <math.h>
3  #include <stdlib.h>
4  #include <omp.h>
5
6  typedef struct
7  {
8    double* ptr;
9  } TILE;
10
11 void
12 MATMUL(TILE* C, TILE* A, TILE* B, int m)
13 {
14   double* a = (double*)A->ptr;
15   double* b = (double*)B->ptr;
16   double* c = (double*)C->ptr;
17
18   for (int i = 0; i < m; i++)
19     for (int j = 0; j < m; j++)
20       for (int k = 0; k < m; k++)
21         c[i * m + j] += a[i * m + k] * b[k * m + j];
22 }
23
24 int
25 main(int argc, char** argv)
26 {
27   assert(argc == 3);
28
29   int TILES = atoi(argv[1]);
30   int nEntries = atoi(argv[2]);
31   long int MATRIX_WIDTH = TILES * nEntries;
32   long int MATRIX_SIZE = MATRIX_WIDTH * MATRIX_WIDTH;
33
34   double* A = (double*)malloc(sizeof(double) * MATRIX_SIZE);
35   double* B = (double*)malloc(sizeof(double) * MATRIX_SIZE);
36   double* R1 = (double*)malloc(sizeof(double) * MATRIX_SIZE);
37   double* R2 = (double*)malloc(sizeof(double) * MATRIX_SIZE);
38
39 #pragma omp parallel for
40   for (int i = 0; i < MATRIX_WIDTH; i++)
41     for (int j = 0; j < MATRIX_WIDTH; j++) {
42       A[i * MATRIX_WIDTH + j] = (rand() % 100) / 100.0;
43       B[i * MATRIX_WIDTH + j] = (rand() % 10) / 10.0;
44       R1[i * MATRIX_WIDTH + j] = 0;
45       R2[i * MATRIX_WIDTH + j] = 0;
46     }
47
48   // initialize tiled matrices
49   TILE* MA = (TILE*)malloc(sizeof(TILE) * TILES * TILES);
50   TILE* MB = (TILE*)malloc(sizeof(TILE) * TILES * TILES);
51   TILE* MC = (TILE*)malloc(sizeof(TILE) * TILES * TILES);
```

```
52
53 #pragma omp parallel for
54   for (int i = 0; i < TILES; i++) {
55     for (int j = 0; j < TILES; j++) {
56       double* a_ptr = (double*)malloc(sizeof(double) * nEntries * nEntries);
57       double* b_ptr = (double*)malloc(sizeof(double) * nEntries * nEntries);
58       double* c_ptr = (double*)malloc(sizeof(double) * nEntries * nEntries);
59       for (int ii = 0; ii < nEntries; ii++)
60         for (int jj = 0; jj < nEntries; jj++) {
61           a_ptr[ii * nEntries + jj] =
62             A[(i * nEntries + ii) * MATRIX_WIDTH + (j * nEntries + jj)];
63           b_ptr[ii * nEntries + jj] =
64             B[(i * nEntries + ii) * MATRIX_WIDTH + (j * nEntries + jj)];
65           c_ptr[ii * nEntries + jj] = 0;
66         }
67       MA[i * TILES + j].ptr = a_ptr;
68       MB[i * TILES + j].ptr = b_ptr;
69       MC[i * TILES + j].ptr = c_ptr;
70     }
71   }
72
73   // matrix multiplication
74   timerstart(&startS, &startNS);
75
76   for (int k = 0; k < TILES; k++)
77 #pragma omp parallel for schedule(runtime)
78     for (int ii = 0; ii < TILES * TILES; ii++) {
79       int i = ii / TILES;
80       int j = ii % TILES;
81       MATMUL(&MC[i * TILES + j], &MA[i * TILES + k], &MB[k * TILES + j],
82              nEntries);
83     }
84   return 0;
85 }
```

Appendix B

Tiled Cholesky Factorization Full Programs

The OCR tiled Cholesky factorization program listed here is adapted from the an example in the public repository of Traleika Glacier XStack Project [17].

Listing B.1: Pure OCR tiled Cholesky factorization implementation

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <malloc.h>
5 #include <math.h>
6 #include <mkl.h>
7 #include <mkl_lapacke.h>
8
9 ocrGuid_t wrapUpTask(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]);
10
11 char uplo = 'L';
12 int nBlocks, nEntries;
13 int lnt, lmt;
14 char prcs; // precision
15 int mThreads;
16 int info;
17
18 double* AMatrix;
19 double* sAMatrix;
20 ocrGuid_t** tiledMatrix = NULL;
21 double*** tiles;
22
23 static inline double
24 random_value_(double a, double b)
25 {
26   return (a > b) ? random_value_(b, a) : (a + ((b - a) * drand48()));
27 }
28
```

```
29 void
30 dpotrf_task_prescriber(ocrGuid_t edtTemplate, u32 k, u32 tileSize,
31                        ocrGuid_t*** depArray, u64 priority)
32 {
33   ocrGuid_t task_guid;
34   u64 paramv[3];
35   paramv[0] = k;
36   paramv[1] = tileSize;
37   paramv[2] = (u64)depArray[k][k][k + 1].guid;
38
39   ocrGuid_t depv[1];
40   depv[0] = depArray[k][k][k];
41
42   ocrHint_t priority_hint;
43   ocrHintInit(&priority_hint, OCR_HINT_EDT_T);
44   ocrSetHintValue(&priority_hint, OCR_HINT_EDT_PRIORITY, priority);
45   ocrEdtCreate(&task_guid, edtTemplate, 3, paramv, 1, depv, EVT_PROP_NONE,
46                &priority_hint, NULL);
47 }
48
49 void
50 dtrsm_task_prescriber(ocrGuid_t edtTemplate, u32 k, u32 i, u32 tileSize,
51                       ocrGuid_t*** depArray, u64 priority)
52 {
53   ocrGuid_t task_guid;
54   u64 paramv[4];
55   paramv[0] = k;
56   paramv[1] = i;
57   paramv[2] = tileSize;
58   paramv[3] = (u64)depArray[i][k][k + 1].guid;
59
60   ocrGuid_t depv[2];
61   depv[0] = depArray[k][k][k + 1];
62   depv[1] = depArray[i][k][k];
63
64   ocrHint_t priority_hint;
65   ocrHintInit(&priority_hint, OCR_HINT_EDT_T);
66   ocrSetHintValue(&priority_hint, OCR_HINT_EDT_PRIORITY, priority);
67   ocrEdtCreate(&task_guid, edtTemplate, 4, paramv, 2, depv, EVT_PROP_NONE,
68                &priority_hint, NULL);
69 }
70
71 void
72 dsyrk_task_prescriber(ocrGuid_t edtTemplate, u32 k, u32 i, u32 tileSize,
73                       ocrGuid_t*** depArray, u64 priority)
74 {
75   ocrGuid_t task_guid;
76   u64 paramv[4];
77   paramv[0] = k;
78   paramv[1] = i;
79   paramv[2] = tileSize;
80   paramv[3] = (u64)depArray[i][i][k + 1].guid;
81
82   ocrGuid_t depv[2];
83   depv[0] = depArray[i][i][k];
84   depv[1] = depArray[i][k][k + 1];
85
86   ocrHint_t priority_hint;
87   ocrHintInit(&priority_hint, OCR_HINT_EDT_T);
88   ocrSetHintValue(&priority_hint, OCR_HINT_EDT_PRIORITY, priority);
89   ocrEdtCreate(&task_guid, edtTemplate, 4, paramv, 2, depv, EVT_PROP_NONE,
90                &priority_hint, NULL);
91 }
92
93 void
94 dgemm_task_prescriber(ocrGuid_t edtTemplate, u32 k, u32 i, u32 j, u32 tileSize,
95                       ocrGuid_t*** depArray, u64 priority)
96 {
97   ocrGuid_t task_guid;
98   u64 paramv[5];
99   paramv[0] = k;
```

```
100   paramv[1] = i;
101   paramv[2] = j;
102   paramv[3] = tileSize;
103   paramv[4] = (u64)depArray[i][j][k + 1].guid;
104
105   ocrGuid_t depv[3];
106   depv[0] = depArray[i][j][k];
107   depv[1] = depArray[i][k][k + 1];
108   depv[2] = depArray[j][k][k + 1];
109
110   ocrHint_t priority_hint;
111   ocrHintInit(&priority_hint, OCR_HINT_EDT_T);
112   ocrSetHintValue(&priority_hint, OCR_HINT_EDT_PRIORITY, priority);
113   ocrEdtCreate(&task_guid, edtTemplate, 5, paramv, 3, depv, EVT_PROP_NONE,
114                &priority_hint, NULL);
115 }
116
117 ocrGuid_t
118 dpotrfTask(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[])
119 {
120   int k = paramv[0];
121   int mRowInTile = (int)paramv[1];
122   ocrGuid_t output;
123   output.guid = paramv[2];
124   double* adata = (double*)depv[0].ptr;
125   LAPACKE_dpotrf(LAPACK_ROW_MAJOR, uplo, mRowInTile, adata, mRowInTile);
126
127   ocrEventSatisfy(output, depv[0].guid);
128   return NULL_GUID;
129 }
130
131 ocrGuid_t
132 dtrsmTask(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[])
133 {
134   int k = paramv[0];
135   int i = paramv[1];
136   int mRowInTile = (int)paramv[2];
137   ocrGuid_t output;
138   output.guid = paramv[3];
139   double* adata = (double*)depv[0].ptr;
140   double* bbdata = (double*)depv[1].ptr;
141   cblas_dtrsm(CblasRowMajor, CblasRight, CblasLower, CblasTrans, CblasNonUnit,
142               mRowInTile, mRowInTile, 1.0, adata, mRowInTile, bbdata,
143               mRowInTile);
144   ocrEventSatisfy(output, depv[1].guid);
145   return NULL_GUID;
146 }
147
148 ocrGuid_t
149 dsyrkTask(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[])
150 {
151   int k = paramv[0];
152   int i = paramv[1];
153   int mRowInTile = (int)paramv[2];
154   ocrGuid_t output;
155   output.guid = paramv[3];
156   double* aadata = (double*)depv[0].ptr;
157   double* bbdata = (double*)depv[1].ptr;
158   cblas_dsyrk(CblasRowMajor, CblasLower, CblasNoTrans, mRowInTile, mRowInTile,
159               -1.0, bbdata, mRowInTile, 1.0, aadata, mRowInTile);
160   ocrEventSatisfy(output, depv[0].guid);
161   return NULL_GUID;
162 }
163 ocrGuid_t
164 dgemmTask(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[])
165 {
166   int k = paramv[0];
167   int i = paramv[1];
168   int j = paramv[2];
169   int mRowInTile = (int)paramv[3];
170   ocrGuid_t output;
```

```
171   output.guid = paramv[4];
172   double* cdata = (double*)depv[0].ptr;
173   double* bcdata = (double*)depv[1].ptr;
174   double* acdata = (double*)depv[2].ptr;
175   cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasTrans, mRowInTile, mRowInTile,
176               mRowInTile, -1.0, bcdata, mRowInTile, acdata, mRowInTile, 1.0,
177               cdata, mRowInTile);
178   ocrEventSatisfy(output, depv[0].guid);
179   return NULL_GUID;
180 }
181
182 ocrGuid_t
183 mainLoopTask(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[])
184 {
185   int nBlocks = paramv[0];
186   int nEntries = paramv[1];
187
188   ocrGuid_t*** dependenceArray;
189   ocrGuid_t tmp_guid;
190
191   ocrDbCreate(&tmp_guid, (void**)&dependenceArray,
192               sizeof(ocrGuid_t**) * nBlocks, DB_PROP_NONE, NULL_HINT, NO_ALLOC);
193   for (int i = 0; i < nBlocks; i++) {
194     ocrDbCreate(&tmp_guid, (void**)&dependenceArray[i],
195                 sizeof(ocrGuid_t*) * (nBlocks), DB_PROP_NONE, NULL_HINT,
196                 NO_ALLOC);
197     for (int j = 0; j <= i; j++) {
198       ocrDbCreate(&tmp_guid, (void**)&dependenceArray[i][j],
199                   sizeof(ocrGuid_t) * (nBlocks + 1), DB_PROP_NONE, NULL_HINT,
200                   NO_ALLOC);
201       for (int k = 0; k <= nBlocks; k++) {
202         ocrEventCreate(&dependenceArray[i][j][k], OCR_EVENT_STICKY_T,
203                        EVT_PROP_NONE);
204       }
205     }
206   }
207
208   // Create task templates
209   ocrGuid_t templateDpotrf, templateDtrsm, templateDsyrk, templateDgemm;
210   ocrEdtTemplateCreate(&templateDpotrf, dpotrfTask, 3, 1);
211   ocrEdtTemplateCreate(&templateDtrsm, dtrsmTask, 4, 2);
212   ocrEdtTemplateCreate(&templateDsyrk, dsyrkTask, 4, 2);
213   ocrEdtTemplateCreate(&templateDgemm, dgemmTask, 5, 3);
214
215   // 3 level loop to create tasks
216   // priority: column index, iteration count, row index
217   u64 max_priority = nBlocks * nBlocks * nBlocks;
218   for (int k = 0; k < nBlocks; k++) {
219     dpotrf_task_prescriber(templateDpotrf, k, nEntries, dependenceArray,
220                            max_priority -
221                              (k * nBlocks * nBlocks + k * nBlocks + k));
222     for (int i = k + 1; i < nBlocks; i++) {
223       dtrsm_task_prescriber(templateDtrsm, k, i, nEntries, dependenceArray,
224                             max_priority -
225                               (k * nBlocks * nBlocks + k * nBlocks + i));
226     }
227     for (int j = k + 1; j < nBlocks; j++) {
228       dsyrk_task_prescriber(templateDsyrk, k, j, nEntries, dependenceArray,
229                             max_priority -
230                               (j * nBlocks * nBlocks + k * nBlocks + j));
231       for (int i = j + 1; i < nBlocks; i++) {
232         dgemm_task_prescriber(templateDgemm, k, i, j, nEntries, dependenceArray,
233                               max_priority -
234                                 (j * nBlocks * nBlocks + k * nBlocks + i));
235       }
236     }
237   }
238
239   for (int i = nBlocks - 1; i >= 0; i--)
240     for (int j = 0; j <= i; j++)
241       ocrEventSatisfy(dependenceArray[i][j][0], tiledMatrix[i][j]);
```

```
242   return NULL_GUID;
243 }
244
245 ocrGuid_t
246 wrapUpTask(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[])
247 {
248   ocrShutdown();
249   return NULL_GUID;
250 }
251
252 ocrGuid_t
253 mainEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[])
254 {
255   int argc = getArgc(depv[0].ptr);
256   char** argv = (char**)malloc(argc * sizeof(char*));
257   for (int i = 0; i < argc; i++) {
258     char* arg = getArgv(depv[0].ptr, i);
259     argv[i] = arg;
260   }
261   /* We will use the same number of entries for each block */
262   if (argc < 4) {
263     fprintf(stderr, "Format: %s S|D #blocks #elements #numThreads\n", *argv);
264     exit(EXIT_FAILURE);
265   }
266
267   /* Read params */
268   prcs = *(argv[1]);
269   nBlocks = strtoul(argv[2], NULL, 10);
270   nEntries = strtoul(argv[3], NULL, 10);
271   mThreads = strtoul(argv[4], NULL, 10);
272
273   mkl_set_num_threads(1);
274
275   // Initialize the matrix
276   lmt = nBlocks * nEntries;
277   lnt = lmt;
278   sAMatrix = memalign(32, lmt * lnt * sizeof(double));
279   double* tptr = sAMatrix;
280   for (int i = 0; i < lmt; ++i) {
281     for (int j = 0; j < i; ++j) {
282       double x = random_value_(-1.0, 1.0);
283       tptr[j + i * lmt] = tptr[i + j * lmt] = x;
284     }
285     tptr[i + i * lmt] = lmt + 1;
286   }
287   AMatrix = memalign(32, lmt * lnt * sizeof(double));
288   memcpy(AMatrix, sAMatrix, lmt * lnt * sizeof(double));
289
290   // Initialize Tiles sequentially
291   ocrGuid_t tmp_guid;
292   ocrDbCreate(&tmp_guid, (void**)&tiledMatrix, sizeof(ocrGuid_t*) * nBlocks,
293               DB_PROP_NONE, NULL_HINT, NO_ALLOC);
294   tiles = (double***)malloc(sizeof(double**) * nBlocks);
295   for (int i = 0; i < nBlocks; i++) {
296     ocrDbCreate(&tmp_guid, (void**)&tiledMatrix[i],
297                 sizeof(ocrGuid_t) * (nBlocks), DB_PROP_NONE, NULL_HINT,
298                 NO_ALLOC);
299     tiles[i] = (double**)malloc(sizeof(double*) * (nBlocks));
300     for (int j = 0; j <= i; j++) {
301       ocrDbCreate(&tiledMatrix[i][j], (void**)&tiles[i][j],
302                   sizeof(double) * nEntries * nEntries, DB_PROP_NONE, NULL_HINT,
303                   NO_ALLOC);
304       for (int x = 0; x < nEntries; x++)
305         for (int y = 0; y < nEntries; y++) {
306           tiles[i][j][x * nEntries + y] =
307             sAMatrix[(i * nEntries + x) * (lmt) + (j * nEntries + y)];
308         }
309     }
310   }
311
312   // Create a finish EDT that spawns tasks to perform cholesky factorization
```

```
313   ocrGuid_t templateMainLoop;
314   ocrGuid_t mainLoopTask_guid;
315   ocrGuid_t mainLoopFinish;
316   u32 new_paramc = 3;
317   u64 new_paramv[new_paramc];
318   new_paramv[0] = nBlocks;
319   new_paramv[1] = nEntries;
320   new_paramv[2] = (u64)tiledMatrix;
321 
322   u32 new_depc = 1;
323   ocrGuid_t new_depv[new_depc];
324   new_depv[0] = UNINITIALIZED_GUID;
325 
326   ocrEdtTemplateCreate(&templateMainLoop, mainLoopTask, new_paramc, new_depc);
327 
328   ocrEdtCreate(&mainLoopTask_guid, templateMainLoop, EDT_PARAM_DEF, new_paramv,
329                EDT_PARAM_DEF, new_depv, EDT_PROP_FINISH, NULL_HINT,
330                &mainLoopFinish);
331 
332   ocrGuid_t templateWrapUp;
333   ocrGuid_t wrapUp;
334 
335   new_paramc = 0;
336   new_depc = 1;
337   new_depv[0] = mainLoopFinish;
338   ocrEdtTemplateCreate(&templateWrapUp, wrapUpTask, new_paramc, new_depc);
339   ocrEdtCreate(&wrapUp, templateWrapUp, EDT_PARAM_DEF, NULL, EDT_PARAM_DEF,
340                new_depv, EDT_PROP_NONE, NULL_HINT, NULL);
341 
342   ocrAddDependence(NULL_GUID, mainLoopTask_guid, 0, DB_MODE_RW);
343   return NULL_GUID;
344 }
```

Listing B.2: HTA-OCR tiled Cholesky factorization implementation

```
1  #include <time.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <malloc.h>
6  #include <math.h>
7  #include <unistd.h>
8  #include <mkl.h>
9  #include <mkl_lapacke.h>
10 
11 #include "HTA.h"
12 #include "HTA_operations.h"
13 #include "Tuple.h"
14 
15 char uplo = 'L';
16 MKL_INT nBlocks, nEntries;
17 int lnt, lmt;
18 char prcs; // precision
19 int mThreads;
20 MKL_INT info;
21 DIST_TYPE dist_type = DIST_ROW_CYCLIC;
22 
23 double* sAMatrix;
24 
25 static inline double
26 random_value_(double a, double b)
27 {
28   return (a > b) ? random_value_(b, a) : (a + ((b - a) * drand48()));
29 }
30 
31 void
32 TRSM(HTA** lhs, HTA** rhs, uint64_t* capture)
```

```
33 {
34   HTA *hAik = lhs[0], hAkk = rhs[0];
35   double* adata = HTA_get_ptr_raw_data(hAkk);
36   double* bbdata = HTA_get_ptr_raw_data(hAik);
37   int mRowInTile = hAkk->flat_size.values[1];
38   cblas_dtrsm(CblasRowMajor, CblasRight, CblasLower, CblasTrans, CblasNonUnit,
39               mRowInTile, mRowInTile, 1.0, adata, mRowInTile, bbdata,
40               mRowInTile);
41 }
42
43 void
44 SYRK(HTA** lhs, HTA** rhs, uint64_t* capture)
45 {
46   HTA *hAij = lhs[0], hAik = rhs[0];
47   int i = hAij->nd_rank.values[1];
48   int mRowInTile = hAij->flat_size.values[1];
49   double* aadata = HTA_get_ptr_raw_data(hAij);
50   double* bbdata = HTA_get_ptr_raw_data(hAik);
51   cblas_dsyrk(CblasRowMajor, CblasLower, CblasNoTrans, mRowInTile, mRowInTile,
52               -1.0, bbdata, mRowInTile, 1.0, aadata, mRowInTile);
53 }
54
55 void
56 GEMM(HTA** lhs, HTA** rhs, uint64_t* capture)
57 {
58   HTA *hAij = lhs[0], hAik = rhs[0], hAjk = rhs[1];
59   int i = hAij->nd_rank.values[1];
60   int j = hAij->nd_rank.values[0];
61   int mRowInTile = hAij->flat_size.values[1];
62   double* cdata = HTA_get_ptr_raw_data(hAij);
63   double* acdata = HTA_get_ptr_raw_data(hAjk);
64   double* bcdata = HTA_get_ptr_raw_data(hAik);
65   cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasTrans, mRowInTile, mRowInTile,
66               mRowInTile, -1.0, bcdata, mRowInTile, acdata, mRowInTile, 1.0,
67               cdata, mRowInTile);
68 }
69
70 void
71 POTRF(HTA** lhs, HTA** rhs, uint64_t* capture)
72 {
73   HTA* hAkk = lhs[0];
74   double* adata = HTA_get_ptr_raw_data(hAkk);
75   int mRowInTile = hAkk->flat_size.values[1];
76   LAPACKE_dpotrf(LAPACK_ROW_MAJOR, uplo, mRowInTile, HTA_get_ptr_raw_data(hAkk),
77                  mRowInTile);
78 }
79
80 int
81 hta_main(int argc, char** argv)
82 {
83   int pid = MYRANK;
84
85   /* We will use the same number of entries for each block */
86   if (argc < 5) {
87     fprintf(stderr, "Format: %s S|D #blocks #elements #numThreads #dist\n",
88            *argv);
89     exit(EXIT_FAILURE);
90   }
91
92   /* Read params */
93   prcs = *(argv[1]);
94   nBlocks = strtoul(argv[2], NULL, 10);
95   nEntries = strtoul(argv[3], NULL, 10);
96   mThreads = strtoul(argv[4], NULL, 10);
97   if (argc > 5) {
98     dist_type = strtoul(argv[5], NULL, 10);
99     assert(dist_type >= 0 && dist_type < DIST_TYPE_MAX);
100  }
101  printf("precision: %c, nBlocks = %d, nEntries = %d, mThreads = %d, dist_type "
102         "= %d\n",
103         prcs, nBlocks, nEntries, mThreads, dist_type);
```

```
104
105   mkl_set_num_threads(1);
106
107   // Initialize the matrix
108   lmt = nBlocks * nEntries;
109   lnt = lmt;
110   printf("Allocating flat matrix of size %dx%d\n", lmt, lnt);
111   sAMatrix = memalign(32, lmt * lnt * sizeof(double));
112   double* tptr = sAMatrix;
113   for (int i = 0; i < lmt; ++i) {
114     for (int j = 0; j < i; ++j) {
115       double x = random_value_(-1.0, 1.0);
116       tptr[j + i * lmt] = tptr[i + j * lmt] = x;
117     }
118     tptr[i + i * lmt] = lmt + 1;
119   }
120   double* AMatrix = memalign(32, lmt * lnt * sizeof(double));
121
122   HTA_SCALAR_TYPE stype = ((prcs == 's') || (prcs == 'S'))
123                             ? HTA_SCALAR_TYPE_FLOAT
124                             : HTA_SCALAR_TYPE_DOUBLE;
125   Tuple t0 = Tuple_create(2, nBlocks, nBlocks);
126   Tuple flat_size = Tuple_create(2, nBlocks * nEntries, nBlocks * nEntries);
127   Tuple mesh = HTA_get_vp_mesh(2);
128   Dist dist;
129   Dist_init(&dist, dist_type, &mesh);
130   HTA* hA =
131     HTA_create_impl(pid, NULL, 2, 2, &flat_size, 0, &dist, stype, 1, &t0);
132
133   // Priority space dimension: (nBlocks * nBlocks * 4)
134   // Priority : -(column index, iteration count, row index)
135   for (int k = 0; k < nBlocks; k++) {
136     Tuple kk = Tuple_create(2, k, k);
137     HTA* hAkk = HTA_pick_one_tile(hA, &kk);
138     HTAOCR_SET_PRIORITY(HTAOCR_EDT_DEFAULT_PRIORITY -
139                         (k * nBlocks * 4 + k * 4 + k));
140     HTA_map(POTRF, ARGS(LHS(0, hAkk)));
141     for (int i = k + 1; i < nBlocks; i++) {
142       Tuple ik = Tuple_create(2, i, k);
143       HTA* hAik = HTA_pick_one_tile(hA, &ik);
144       HTAOCR_SET_PRIORITY(HTAOCR_EDT_DEFAULT_PRIORITY -
145                           (k * nBlocks * 4 + k * 4 + i));
146       HTA_map(TRSM, ARGS(LHS(0, hAik), RHS(0, hAkk)));
147     }
148
149     double iter_count = k;
150     for (int j = k + 1; j < nBlocks; j++) {
151       Tuple jj = Tuple_create(2, j, j);
152       Tuple jk = Tuple_create(2, j, k);
153       HTA* hAjk = HTA_pick_one_tile(hA, &jk);
154       HTA* hAjj = HTA_pick_one_tile(hA, &jj);
155       HTAOCR_SET_PRIORITY(HTAOCR_EDT_DEFAULT_PRIORITY -
156                           (j * nBlocks * 4 + k * 4 + j));
157       HTA_map(SYRK, ARGS(LHS(0, hAjj), RHS(0, hAjk)));
158       for (int i = j + 1; i < nBlocks; i++) {
159         Tuple ij = Tuple_create(2, i, j);
160         Tuple ik = Tuple_create(2, i, k);
161         HTA* hAij = HTA_pick_one_tile(hA, &ij);
162         HTA* hAik = HTA_pick_one_tile(hA, &ik);
163         HTAOCR_SET_PRIORITY(HTAOCR_EDT_DEFAULT_PRIORITY -
164                             (j * nBlocks * 4 + k * 4 + i));
165         HTA_map(GEMM, ARGS(LHS(0, hAij), RHS(0, hAik, hAjk)));
166       }
167     }
168   }
169
170   Tuple last = Tuple_create(2, nBlocks - 1, nBlocks - 1);
171   HTA* hAkk = HTA_pick_one_tile(hA, &last);
172   HTA_UNCLAIM_RET(1, hAkk);
173
174   return 0;
```

```
175 }
```

Listing B.3: Pure OpenMP tiled Cholesky factorization implementation

```
1 #include <stdio.h>
2 #include <math.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <malloc.h>
6 #include <omp.h>
7 #include <mkl.h>
8 #include <mkl_lapacke.h>
9 
10 char uplo = 'L';
11 int lnt, lmt;
12 MKL_INT nBlocks, nEntries;
13 char prcs; // precision
14 int mThreads;
15 double* sAMatrix;
16 
17 typedef struct
18 {
19   int n;
20   double* val;
21 } TILE;
22 
23 static inline double
24 random_value_(double a, double b)
25 {
26   return (a > b) ? random_value_(b, a) : (a + ((b - a) * drand48()));
27 }
28 
29 void
30 GET_I_J(int x, int start, int end, int* ret_i, int* ret_j)
31 {
32   int i = start;
33   while (i < end) {
34     if (x < end - i)
35       break;
36     x -= end - i;
37     i++;
38   }
39   *ret_i = x + i;
40   *ret_j = i;
41 }
42 
43 TILE*
44 allocate_tiled_matrix(int nBlocks, int nEntries)
45 {
46   TILE* A = (TILE*)malloc(sizeof(TILE) * nBlocks * nBlocks);
47   for (int i = 0; i < nBlocks * nBlocks; i++) {
48     A[i].n = nEntries;
49     A[i].val = (double*)malloc(sizeof(double) * nEntries * nEntries);
50   }
51   return A;
52 }
53 
54 void
55 initialize_tiled_matrix(TILE* A, double* AMatrix, int nBlocks, int nEntries)
56 {
57 #pragma omp parallel
58   {
59 #pragma omp for
60     for (int i = 0; i < nBlocks; i++)
61       for (int j = 0; j < nBlocks; j++) {
62         TILE* t = &A[i * nBlocks + j];
63         for (int ii = 0; ii < nEntries; ii++)
```

```
64          for (int jj = 0; jj < nEntries; jj++) {
65            t->val[ii * nEntries + jj] =
66              AMatrix[(i * nEntries + ii) * nBlocks * nEntries +
67                      (j * nEntries + jj)];
68          }
69      }
70  }
71 }
72
73 void
74 POTRF(TILE* t)
75 {
76   double* adata = t->val;
77   int mRowInTile = t->n;
78   LAPACKE_dpotrf(LAPACK_ROW_MAJOR, uplo, mRowInTile, adata, mRowInTile);
79 }
80
81 void
82 TRSM(TILE* Aik, TILE* Akk)
83 {
84   double* adata = Akk->val;
85   double* bbdata = Aik->val;
86   int mRowInTile = Akk->n;
87   cblas_dtrsm(CblasRowMajor, CblasRight, CblasLower, CblasTrans, CblasNonUnit,
88               mRowInTile, mRowInTile, 1.0, adata, mRowInTile, bbdata,
89               mRowInTile);
90 }
91
92 void
93 SYRK(TILE* Aij, TILE* Aik)
94 {
95   int mRowInTile = Aij->n;
96   double* aadata = Aij->val;
97   double* bbdata = Aik->val;
98   cblas_dsyrk(CblasRowMajor, CblasLower, CblasNoTrans, mRowInTile, mRowInTile,
99               -1.0, bbdata, mRowInTile, 1.0, aadata, mRowInTile);
100 }
101
102 void
103 GEMM(TILE* Aij, TILE* Aik, TILE* Ajk)
104 {
105   int mRowInTile = Aij->n;
106   double* cdata = Aij->val;
107   double* acdata = Ajk->val;
108   double* bcdata = Aik->val;
109   cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasTrans, mRowInTile, mRowInTile,
110               mRowInTile, -1.0, bcdata, mRowInTile, acdata, mRowInTile, 1.0,
111               cdata, mRowInTile);
112 }
113
114 int
115 main(int argc, char** argv)
116 {
117   mkl_set_num_threads(1);
118
119   if (argc < 4) {
120     fprintf(stderr, "Format: %s S|D #blocks #elements\n", *argv);
121     exit(EXIT_FAILURE);
122   }
123
124   fprintf(stderr, "Initializing...");
125   fflush(stderr);
126
127   /* Read params */
128   prcs = *(argv[1]);
129   nBlocks = strtoul(argv[2], NULL, 10);
130   nEntries = strtoul(argv[3], NULL, 10);
131 #pragma omp parallel
132   mThreads = omp_get_num_threads();
133
134   // Initialize the matrix
```

```
135   lmt = nBlocks * nEntries;
136   lnt = lmt;
137   printf("Allocating flat matrix of size %dx%d\n", lmt, lnt);
138   sAMatrix = memalign(32, lmt * lnt * sizeof(double));
139   double* tptr = sAMatrix;
140   for (int i = 0; i < lmt; ++i) {
141     for (int j = 0; j < i; ++j) {
142       double x = random_value_(-1.0, 1.0);
143       tptr[j + i * lmt] = tptr[i + j * lmt] = x;
144     }
145     tptr[i + i * lmt] = lmt + 1;
146   }
147   double* AMatrix = memalign(32, lmt * lnt * sizeof(double));
148
149   // allocate tiled matrix
150   TILE* A = allocate_tiled_matrix(nBlocks, nEntries);
151
152   // Copy initialized data (parallel)
153   initialize_tiled_matrix(A, sAMatrix, nBlocks, nEntries);
154
155   // cholesky computation
156   for (int k = 0; k < nBlocks; k++) {
157
158     int numDGEMMS = (nBlocks - k) * (nBlocks - k - 1) / 2;
159
160     POTRF(&A[k * nBlocks + k]);
161
162 #pragma omp parallel for schedule(runtime)
163     for (int i = k + 1; i < nBlocks; i++) {
164       TRSM(&A[i * nBlocks + k], &A[k * nBlocks + k]);
165     }
166
167 #pragma omp parallel for schedule(runtime)
168     for (int x = 0; x < numDGEMMS; x++) {
169       int i, j;
170       GET_I_J(x, k + 1, nBlocks, &i, &j);
171       if (i == j)
172         SYRK(&A[j * nBlocks + j], &A[j * nBlocks + k]);
173       else
174         GEMM(&A[i * nBlocks + j], &A[i * nBlocks + k], &A[j * nBlocks + k]);
175     }
176   }
177
178   return 0;
179 }
```

Bibliography

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[2] D. Andrade, B. B. Fraguela, J. Brodman, and D. Padua. Task-parallel versus data-parallel library-based programming in multicore systems. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 101–110. IEEE, 2009.

[3] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[4] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, et al. The nas parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[5] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, S. Kumar, E. Lusk, R. Thakur, and J. L. Träff. Mpi on a million processors. In *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*, pages 20–30. Springer, 2009.

[6] 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*, page 66. IEEE Computer Society Press, 2012.

[7] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzarán, D. Padua, and C. Von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–57. ACM, 2006.

[8] O. A. R. Boards. Openmp specifications. `http://www.openmp.org/specifications/`, 2017. [Online; accessed 04-June-2017].

[9] S. Borkar. Tralieka-glacier final scientific/technical report. Technical report, Intel Federal LLC, 2015.

[10] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. Dague: A generic distributed dag engine for high performance computing. *Parallel Computing*, 38(1):37–51, 2012.

[11] Z. Budimlic, V. Cavé, S. Chatterjee, R. Cledat, V. Sarkar, B. Seshasayee, R. Surendran, and N. Vrvilo. 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. Austin, Texas, November 2015*, 2015.

[12] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53, 2009.

[13] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. Ieee, 2009.

[14] T. Consortium. Teraflux applications. `https://svn.teraflux.eu/svnpub/apps/`, 2017. [Online; accessed 04-June-2017].

[15] D. G. Costa, T. Fahringer, J.-A. Rico-Gallego, I. Grasso, A. Hristov, H. D. Karatza, A. Lastovetsky, F. Marozzo, D. Petcu, G. L. Stavrinides, et al. Exascale machines require new programming paradigms and runtimes. *Supercomputing frontiers and innovations*, 2(2):6–27, 2015.

[16] A. Duran, J. M. Perez, E. Ayguadé, R. M. Badia, and J. Labarta. Extending the openmp tasking model to allow dependent tasks. In *International Workshop on OpenMP*, pages 111–122. Springer, 2008.

[17] M. Foundation. Traleika glacier project. `https://xstackwiki.modelado.org/Traleika_Glacier`, 2017. [Online; accessed 04-June-2017].

[18] B. B. Fraguela, G. Bikshandi, J. Guo, M. J. Garzarán, D. Padua, and C. Von Praun. Optimization techniques for efficient hta programs. *Parallel Computing*, 38(9):465–484, 2012.

[19] B. B. Fraguela, J. Guo, G. Bikshandi, M. J. Garzaran, G. Almasi, J. Moreira, and D. Padua. The hierarchically tiled arrays programming approach. In *Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems*, pages 1–12. ACM, 2004.

[20] A. Gara, M. A. Blumrich, D. Chen, G.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, et al. Overview of the blue gene/l system architecture. *IBM Journal of Research and Development*, 49(2.3):195–212, 2005.

[21] D. Goodell, W. Gropp, X. Zhao, and R. Thakur. Scalable memory use in mpi: a case study with mpich2. In *European MPI Users' Group Meeting*, pages 140–149. Springer, 2011.

[22] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[23] W. Gropp and M. Snir. Programming for exascale computers. *Computing in Science & Engineering*, 15(6):27–35, 2013.

[24] R. Hornung, J. Keasler, and M. Gokhale. Hydrodynamics challenge problem. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2011.

[25] Intel. Intel math kernel library. `https://software.intel.com/en-us/mkl`, 2017. [Online; accessed 04-June-2017].

[26] H.-Q. Jin, M. Frumkin, and J. Yan. The openmp implementation of nas parallel benchmarks and its performance. 1999.

[27] L. V. Kale and S. Krishnan. *CHARM++: a portable concurrent object oriented system based on C++*, volume 28. ACM, 1993.

[28] I. Karlin, J. Keasler, and J. Neely. Lulesh 2.0 updates and changes. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2013.

[29] L. L. N. Laboratory. Coral benchmarks. `https://asc.llnl.gov/CORAL-benchmarks/`, 2017. [Online; accessed 04-June-2017].

[30] C. Lauderdale, M. Glines, J. Zhao, A. Spiotta, and R. Khan. Swarm: A unified framework for parallel-for, task dataflow, and distributed graph traversal. *ET International Inc., Newark, USA*, 2013.

[31] K. London, S. Moore, P. Mucci, K. Seymour, and R. Luczak. The papi cross-platform interface to hardware performance counters. In *Department of Defense Users Group Conference Proceedings*, pages 18–21, 2001.

[32] T. Mattson, R. Cledat, Z. Budimlic, V. Cave, S. Chatterjee, B. Seshasayee, R. van der Wijngaart, and V. Sarkar. Ocr: The open community runtime interface version 1.1.0, 2015.

[33] T. G. Mattson, R. Cledat, V. Cavé, V. Sarkar, Z. Budimlić, S. Chatterjee, J. Fryman, I. Ganev, R. Knauerhase, M. Lee, et al. The open community runtime: A runtime system for extreme scale computing. In *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, pages 1–7. IEEE, 2016.

[34] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang. Introducing the graph 500. *Cray Users Group (CUG)*, 2010.

[35] J. M. Perez, R. M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Cluster Computing, 2008 IEEE International Conference on*, pages 142–151. IEEE, 2008.

[36] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical task-based programming with starss. *International Journal of High Performance Computing Applications*, 23(3):284–299, 2009.

[37] J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* " O'Reilly Media, Inc.", 2007.

[38] V. Sarkar, W. Harrod, and A. E. Snavely. Software challenges in extreme scale systems. In *Journal of Physics: Conference Series*, volume 180, page 012045. IOP Publishing, 2009.

[39] V. Sarkar and J. Hennessy. Partitioning parallel programs for macro-dataflow. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 202–211. ACM, 1986.

[40] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken. Regent: a high-productivity programming language for hpc with logical regions. In *High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for*, pages 1–12. IEEE, 2015.

[41] A. R. Smith. *The Parallel Intermediate Language*. PhD thesis, University of Illinois at Urbana-Champaign, 2015.

[42] E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer. Top500 list. `https://www.top500.org/`, 2017. [Online; accessed 04-June-2017].

[43] C.-C. Yang, J. C. Pichel, A. R. Smith, and D. A. Padua. Hierarchically tiled array as a high-level abstraction for codelets. In *Data-Flow Execution Models for Extreme Scale Computing (DFM), 2014 Fourth Workshop on*, pages 58–65. IEEE, 2014.

[44] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao. Using a codelet program execution model for exascale machines: position paper. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, pages 64–69. ACM, 2011.