High-level Status Summary

Technology	**Description (Institution)**	**Status**
Applications/Runtime	TCE mapping/porting to OCR block level (ETI)	In Process
Applications/Runtime	TCE mapping/porting to SWARM Function level (ETI)	Done
Memory access semantics/Runtime	Evaluation of memory semantics based on TCE mapping to OCR and SWARM (ETI)	Evaluating
Memory semantics	Composing a position paper on virtual memory model based on codelets and related memory semantics (ETI)	In Progress
Parallelizing Compiler	Unstructured Mesh computations	Evaluating
Parallel Language	HTA library and PIL compiler design and implementation changes targeting the distributed memory environment (UIUC)	In Process
Parallel Language	Evaluation of PIL targeting the distributed memory SCALE with NAS Parallel Benchmarks (UIUC)	Evaluating
Parallel Language	Performance evaluation and overhead analysis of HTA on shared memory machines (UIUC)	In Process
Parallel Language	Irregular tiles support in HTA (UIUC)	Done
Applications	Lulesh refactoring (PNNL)	Done

Enhanced Data Types	Design of Composite Data Types (PNNL)	In Process
Enhanced Data Types	Group Locality (PNNL)	In Process

Summaries of Quarterly Work (Q7)

ETI Work

During this reporting period (Q7: 03/01/2014-05/31/2014), ETI has been working on the following tasks, according to the SOW.

Task 2.2: Research memory access semantics (in progress)

Task 2.3: Research memory movement policies and interface to compiler (in progress)

Task 7.1: Define intermediate representation (in progress)

During this quarter, main progress was made in connection with Task 2.2 and Task 2.3.

We focused on TCE (Tensor Contraction Engine) -- a DoE proxy application identified by PNNL, our co-design partner, to be studied under this proposal. Since the actual research work on TCE involves both Task 2.2 and 2.3 (and task 2.1) -- we organized the reporting work together and the individual progress toward each task should be easily recognized from the context and illustration.

NOTE: Overall, we have studied two DoE proxy apps provided by PNNL: SCF and TCE. We have also summarized what we learned from LULESH as well as other benchmarks in the DoE domain. We are in the process of planning a publication on the preliminary results of tasks 2.1-2.3.

TCE

ETI has continued to adapt the TCE code generator to generate task-based runtime code. In the previous quarter (Q6), we got basic code generation working for the Open-Community Runtime (OCR), and were working on generating equivalent code for the SWARM runtime. Since then, the basic code generation for the SWARM runtime has also been completed.

This basic task-based runtime code gives us a very coarse level of parallelism. The data is represented as one memory buffer per tensor. The parallelism is limited to the number of tensor contractions specified in the input file, with one EDT/codelet per tensor contraction. Those EDTs/codelets vary wildly in execution time, resulting in poor load balance, and poor utilization of processing resources. We have begun to decompose the problem further, to take advantage of the block-sparse tensor data format in use. This will allow us to express parallelism at the block level, with one data block per block of tensor data, and one EDT/codelet for every block-level calculation. An example of this would be a task which takes two blocks from their respective input tensors, multiplies them together, and adds the result to a block of the output tensor.

This will parallelize the critically large (order N^5) tensor contraction expressions which occupy the majority of the execution time, and will allow us to start studying the data placement and data movement characteristics of this application. In particular, it is important to understand how excessively large data structures (too large to fit on a single compute node) should be split across compute nodes, and how to organize the computation and the other data in order to minimize the necessary communication between compute nodes.

As we perform this research, it will also be important to take the execution context into account. In this standalone proxy application, once execution is done, the program is over. However, in the real NWChem application, this code represents a single step of an iterative algorithm, whose output is subjected to further processing, and then the whole thing starts over. If we simply optimize this code without understanding the context, we may distribute the input and output data in such a way that works great for the TCE code itself, but causes additional data movement costs which slow down the subsequent processing in the NWChem code. Therefore, to maximize the usefulness of our results, the subsequent processing should be taken into account and attempts should be made to optimize the application as a whole.

Future Work

- Q8
    - TCE Application
        - Fine-grained parallelism for OCR and SWARM runtimes
        - Distributed operation (SWARM runtime)
        - Performance improvement through smart data placement, data movement and scheduling decisions

    - Collaborate with UIUC to attain good performance results for HTA generated SWARM/SCALE code
- Year 3
    - Applications support reliability via containment domains
    - Migration paths for MPI+OpenMP applications

Reservoir Work

This quarter has been slower than usual, as we had to temporarily give our full attention to a different project. We worked on Task 3.4 - Optimizing unstructured mesh computations.

We have started our effort to parallelize codes for unstructured meshes by studying a subset of them, called “Structured Adaptive Mesh Refinement” (SAMR). In particular, we are looking at the Chombo Framework from Lawrence Berkeley National Labs, and the underlying BoxLib infrastructure as the reference SAMR implementation. Our initial goal is to learn about the algorithms, parallelization strategies and data-structures involved in Chombo and BoxLib.
We also looked at the approach that uses Legion as a task graph programming API.
The main source of unstructuredness in Chombo is the data-dependent position and size of the hierarchy of boxes in which computations occur.

Forward plan

- Q8-Q12
    - Unstructured mesh computations
        - Started, w/ Chombo study.
        - Considering data-centric approach
- Q8-Q10
    - In support of Unstructured mesh computations
        - Data/communication layer (with DynAX team)
        - Keep improving aspects of R-Stream as needed

UIUC Work

In this quarter, the UIUC team worked mainly on two items:

Task 4.2: Irregular tiles (completed in Y2 Q3)

Task 5.3’: Evaluation of the PIL implementation and API (in progress, shared memory version)

First, we extended the HTA library to support irregular tiles partitioning (task 4.2). By having the partitioning mechanism, it is now possible for applications to create irregular tiles and thus providing more flexibility in controlling the granularity of parallel computation tasks. The API functions HTA_part() and HTA_rmpart() are added into the HTA library implementation, and we plan to investigate whether any of the NAS benchmark programs can utilize this feature to improve performance.

Second, we analyzed the benchmarking results of HTA running on ETI's shared memory SCALE (task 5.3). The overhead of parallel operation invocation can come from different layers of the software stack, such as the HTA library, the code generated by PIL compiler, and the runtime system. To find out which layer is the greatest source of overhead, we designed a mini-benchmark program which includes hand-coded SCALE, hand-coded OpenMP, and HTA versions. After comparing the results of these versions, we identified some inefficiencies existing in the automatically generated SCALE code by the PIL compiler and ways to avoid this problem. The improved version shows significant reduction in the overhead. More about this is described in the details section.

Future Work

- Goals for August '14
    - Complete HTA implementation for distributed memory machines
    - Identify the overhead in different levels
        - HTA library
        - Compiler generated code
        - Codelet runtime
    - Complete R-Stream integration
- Goals for August '15
    - Extend HTA design and implementation to include multiple levels of parallelism, irregular parallelism
    - Implement a variety of other benchmarks
        - Mini GMG, AMR, etc
    - Evaluate and tune for performance
    - Evaluate programmability using objective metrics (e.g. number of operations)

PNNL Work

In this quarter, we applied compression/decompression ACDT for sparse matrices to Matrix-Vector-Multiply and Cholesky Decomposition; extended our tiling strategy with intra-tile parallelism to support multi-hierarchical concurrent start; continued development of our framework for fine-grained parallelism; and improved data movement and the restructuring of data.

Future work

- Q8
    - Application of ACDT compression/decompression to the Distributed Cholesky Decomposition.
    - Application of ACDT to other use cases such as improved resiliency with added redundancy (inverse of compression)
- Year 3
    - Potential application of ACDT to OCR.
    - Test different memory mappings on Tilera to orchestrate data movement and avoid memory interference in memory banks and pages.

Topic Detail:

UIUC

Irregular Tiles Support in HTA

Several functions are added to the HTA library implementation in order to support irregular tiles. The users can create irregular HTA from a 2D array, and they can dynamically insert/remove partition lines to change the tiling at run time.

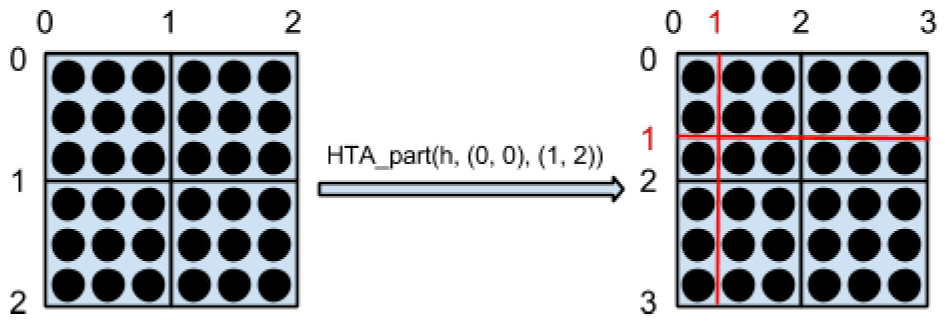
The following code is the data type definition for users to specify the partition lines along a dimension. num_parts indicates the number of partition lines along this dimension, and the number of partitions is (num_parts+1). The values array stores the position of the partition lines. The size of it is statically determined to avoid the overhead of having to use malloc.

```
typedef struct part {
    int num_parts;
    int values[HTA_MAX_NUM_PART];
} Partition;
```

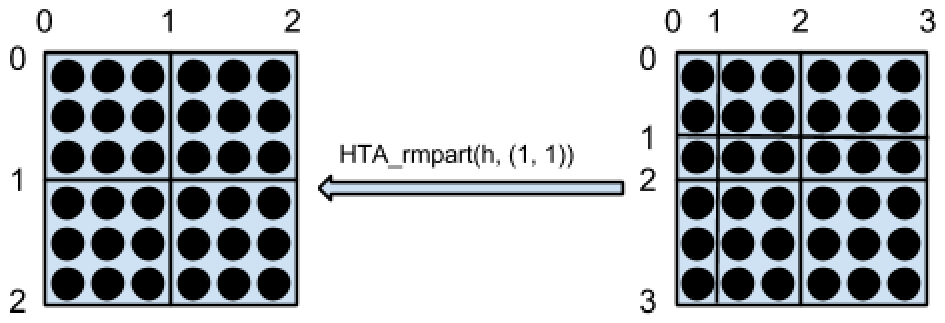
The API functions added are:

```
HTA* HTA_part(HTA* h, const Partition* src_partition, const Partition* offset);
HTA* HTA_rmpart(HTA *h, const Partition* partition);
HTA* HTA_part_matrix(int dim, void *matrix, Tuple* matrix_size, HTA_SCALAR_TYPE
scalar_type, Partition* partitions);
```

HTA_part() lets you add a partition line along each dimension. For example, HTA_part(h, (0, 0), (1, 2)) means to add partition lines at the 0th partition at both dimensions, and to insert the line at offset 1 and 2. The following figure illustrates the effect of performing this operation.



HTA_rmpart() lets you remove the partition lines by specifying the index of the partition line. Notice that the boundary partition lines cannot be removed, so the index has to be less than (num_parts) and larger than zero.



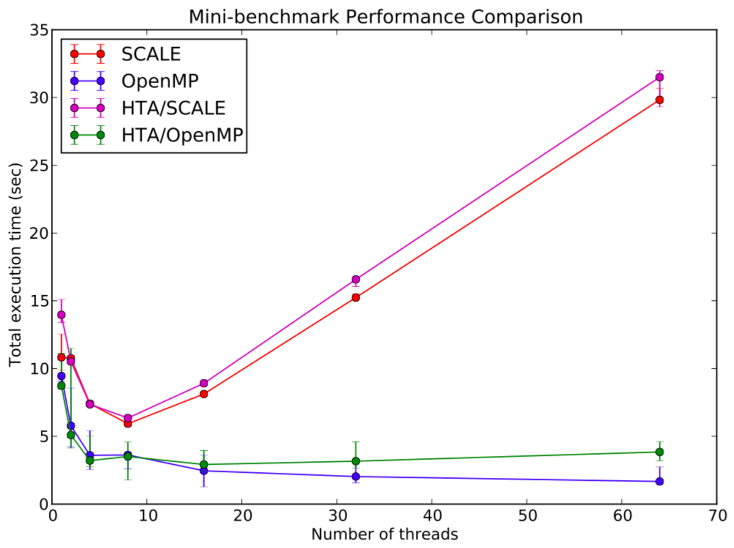
HTA_part_matrix() is an HTA creation operation. It take as input the partitioning and an N-D array. It then allocates space for the HTA and copy the values from the array and returns an HTA object.

With these functions, users can now create irregular HTAs and change the tiling of HTAs at execution time, which enables the possibility to have better control over data distribution and re-distribute data dynamically and both implies more flexibility in load balancing of the computation. Our next step will be to exploit these API functions and improve the performance of existing applications.

Overhead Analysis

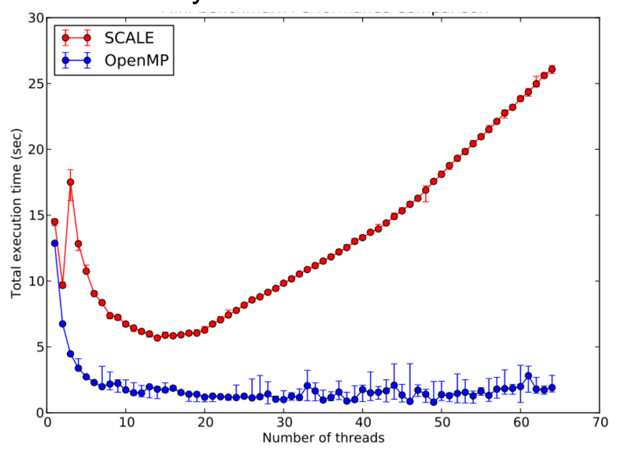
The preliminary performance results of NAS parallel benchmarks implemented in HTA running on SWARM runtime are slower compared with the pure-OpenMP implementation. Because the software stack contains many layers including HTA library, PIL generated SCALE code, and the SWARM runtime, it is not easy to locate the major source of the overhead directly. We decided to create a mini-benchmark program which behaves similarly to the NAS benchmark programs but with simpler computation, and implement the mini-benchmark directly in hand-coded SCALE, hand-coded OpenMP, and compare them with the HTA version.

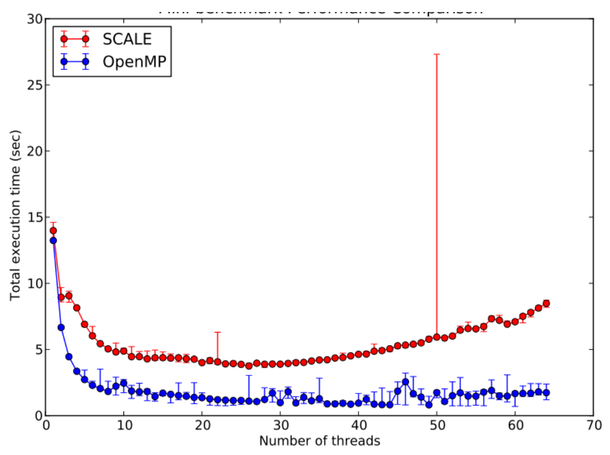
The mini-benchmark is an application that performs a large number of parallel operation invocation within a sequential for loop. The parallel operation takes two tiled arrays, and each parallel task takes one tile from each array, performs element-to-element multiplication and stores the result to another tile. It is embarrassingly parallel and memory bound. The problem size is fixed and we observe the strong scaling effects in the experiments.



The above figure shows the performance results of four different implementations of the mini-benchmark program. As can be seen, the hand-coded SCALE version and the HTA-to-SCALE version both show linear growth in the execution time when the number of threads used increases.

We performed detailed analysis with the ETI team and found that the overhead is due to the sequential spawning of parallel codelets which requires argument boxing. By using swarm_Locale_scheduleToLeaves() which assign work for each of the available worker thread without argument boxing, the overhead is reduced significantly. Right now, we are working on implementing this change in the PIL compiler in order to see the effects on performance of NAS Parallel Benchmarks. We will keep exploring the chances in reducing the overhead in different levels and our goal is to get the performance HTA-to-SCALE version close to that of OpenMP on shared memory machines.





(a) Invoking parallel tasks in a for loop (b) Invoking tasks using swarm_Locale_scheduleToLeaves()

PNNL

Under the Power-Efficient Data Abstraction Layer (PEDAL) subproject we implemented a version of Rescinded Primitive Data Type Access (RPTDA) on SWARM. The approach is a bottom-up encapsulation of vectors, matrices and structures only visible to the RTS. These encapsulated data types which we call Architected Composite Data Types (ACDT) are associated with data transformation layouts and specific operators. We applied compression/decompression ACDT for sparse matrices to Matrix-Vector-Multiply and Cholesky Decomposition. The first variant shows promise with HW support whereas the second variant, solely implemented in SW, already yields improvements: With minimal changes to the hand-optimized Cholesky Decomposition code to support ACDT, we measured for a representative set of sparse matrices ~10x performance improvements AND ~30% power efficiency improvements vs. the default implementation.

Group locality is a concept in which threads collaborate at a very fine grain level. Such collaboration happens from both compute and memory perspective. Processing units work together such that tasks are executed at very fine granularity and synchronize mainly using atomic operation. Similarly, using careful orchestration of memory access, data movement and data restructuring, interference in memory is reduced and accesses are made mostly contiguous.

In this quarter, we extended our highly parallel tiling strategy (Jagged Tiling) with intra-tile parallelism to support multi-hierarchical concurrent start. We continued development of our framework for fine-grained parallelism where threads perform a micro dataflow execution. The framework currently uses PLUTO and CLOOG as a tool for tiled code generation. Finally, we improved data movement and the restructuring of data such that accesses with reuse are preserved in a state that matches the pattern of future access minimizing interference.

We revised and resubmitted the paper, “Architected Composite Data Types: Trading-in Unfettered Data Access for Improved Execution,” to Cluster 2014.
Issues: We are testing jagged tiling to extract both outer and inner tile concurrent start. We are trying to figure out the methodology to automate the code generation process. Guaranteeing access interference between threads in physical memory requires very careful address mapping and access orchestration. Currently with Intel Xeon Phi as our experimental platform, such endeavor has not been successful yet.