High-level Status Summary

Technology	Description (Institution)	Status
Applications/Runtime	TCE mapping/porting to OCR block level (ETI)	In Progress
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)	Done
Parallelizing compiler	Compiler scalability - Fast hidden equality computation	Done
Parallelizing compiler	Parallelizing applications - blocked structured mesh computations.	Evaluating
Parallel Language	HTA library and PIL compiler design and implementation changes for SPMD execution (UIUC)	In Progress
Parallel Language	Evaluation of SPMD mode with NAS Parallel Benchmarks (UIUC)	Evaluating
Parallel Language	Performance evaluation and overhead analysis of HTA on shared memory machines (UIUC)	In Progress
Applications	Lulesh refactoring (PNNL)	Done
Enhanced Data Types	Design of Composite Data Types (PNNL)	In Progress

Summaries of Quarterly Work (Q8)

ETI Work

During this reporting period (Q8: 06/01/2014-08/31/2014), 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.

We have also worked in collaboration with UIUC to analyze and improve the performance of HTA-compiled applications.

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. We are also working to produce a report on the state of our intermediate representation (SCALE).

TCE

ETI has continued to adapt the TCE code generator to generate task-based runtime code. In the previous quarters (Q6 and Q7), we got basic code generation working for the Open-Community Runtime (OCR) and the SWARM runtime. This code gives us a very coarse level of parallelism, with one task per tensor contraction, and about 40 contractions in total (depending on the

correlation model in use). In this quarter, we have worked on exposing more parallelism, and balancing the workload.

We have been working 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. A typical task would take two blocks from their respective input tensors, multiply them together, apply a scalar coefficient, and add the result to a block of the output tensor.

This parallelizes the critically large (order N^5) tensor contraction expressions which occupy the majority of the execution time, and allows us to overlap execution between tensor contractions in order to balance the overall application workload. It will also 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.

Some effort was required to adapt the application's data structures to allow this amount of asynchrony. An alternate representation of tensors and blocks is required. Rather than working on packed arrays and inferring the size and orientation of tensor blocks from global variables or outer loop iterators, the code now has a notion of futures (represented in the OCR version as Event GUIDs), and keeps track of the parameters of a block (size and position) separately from the block itself. Tensor and block metadata is now stored separately from the block data itself, with an efficient means of looking up a future block's metadata within a tensor. This allows the inputs and outputs of tasks to be declared in terms of futures, and the application dependency graph can be planned out and built ahead of time.

This work has also exposed a significant amount of redundant computation, in two cases. In the first case, the input blocks for a block-level contraction may be permuted / transposed, depending on where the block sits within the tensor and the symmetry pattern of the tensor. This permutation / transposition may happen many times, once each time the block is used as input for a contraction task. In the second case, huge tensors are often stored in a compact form, and expanded as they are read. See 2eorb and 2emet for details of this expansion. The block expansion occurs each time the block is read as input for a contraction task. If the resulting data could be reused, without expanding the memory footprint to an unreasonable degree, we believe this would result in a significant application performance boost.

The work adapting the TCE application to task-based runtimes will continue into the next quarter.

HTA Performance

ETI worked with UIUC to improve performance of the HTA library when using SWARM. We found a few points of improvement to the SWARM runtime and the PIL compiler that allowed for better performance on the NAS Parallel Benchmarks. These include improved sleep/wake code in SWARM and more efficient codelet generation in the PIL->SCALE layer. This resulted in a large improvement on the overhead between iterations of the CG benchmark, enough to significantly affect the overall runtime.

Memory model and semantics

We have engaged and pursued the research work on memory model and semantics - based on our research and experience in codelet execution model and its abstract architecture implications. We have reached an initial conclusion: we believe that it is feasible to build a multi-core operating system that implements virtual memory, and honors the principles of modular software construction, using runtime software that executes a codelet program execution model. Performance and energy efficiency can be enhanced through co-design of new architecture features that replace resource management functions of runtime software with efficient hardware mechanisms. The resulting systems will offer benefits in programmability, application portability and reuse absent in current systems.

Our findings are summarized and published in a position paper where we argue that virtual memory is required to achieve the flexibility of resource management demanded for future applications of massively parallel computer systems. Moreover, a virtual memory implementation is a prerequisite for building systems that can support the general composability of parallel programs [2]. We believe the best route to a major improvement in programmability of massively parallel systems is to extend the virtual memory concept to the domain of parallel processing. Then, programmers will be freed from managing processor assignment and scheduling, just as virtual memory freed them from involvement in memory management. In summary, it is feasible to implement a multi-core operating system that will support composability of parallel programs, an achievement that will yield benefits in programmability, ease of reuse and portability, as well as performance and energy efficiency.

Details -- please see Topic Detail for further references.

Future Work

In the next quarter, we plan to continue adapting the TCE application, and to begin researching Resilience (Task 8.1) and MPI Interoperability (Task 10.1).

- Short term (Q9)
    - Continue adapting TCE application

- Begin work on Resilience
    - Begin work on MPI Interoperability
- Longer term (Q10-Q12)
    - Continue work on Resilience
    - Continue work on MPI Interoperability

Reservoir Work

During this reporting period, Reservoir has been working on the following SOW tasks:

Task 3.3: Improve scalability of polyhedral mapping
Task 3.4: Optimize unstructured mesh computations

Compiler scalability

This work is part of our effort to improve the scalability of the polyhedral mapping engine in R-Stream (Task 3.3 of the proposal). We designed and implemented a novel algorithm for computing the set of hidden equalities implied by a set of inequalities representing a polyhedron. This technique is presented in the detail section “Scalable Hidden Equalities Detection” below.

Unstructured mesh computations

We are focusing our application effort on HPGMG, a geometric multi-grid solver, which is one of the building blocks of Block Sparse Adaptive Mesh Refinement codes as used in the ExaCT co-design center. We are using R-Stream to produce a SWARM-parallelized version automatically. In this process, we are also using this effort on a “dusty deck” code as an opportunity to work on reducing the need for a user to rewrite code to a mappable form. While this part of the code is actually structured, it seems to be the most sensible starting point in the process of mapping this block-structured type of application.

Forward plan

- Q9-Q10
    - In support of Unstructured mesh computations
        - Data/communication layer (with DynAX team)
        - Keep improving aspects of R-Stream as needed
- Q11-Q12
    - Unstructured mesh computations
        - Data-centric approach based on the communication layer

UIUC Work

In this quarter, the UIUC team worked mainly on the item:

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

We focused on understanding the causes of overheads in the HTA library, PIL generated code, and potential SWARM runtime system problems. Several optimizations were implemented and the performance of the NAS benchmarks was improved greatly. The details are reported in the later section.

We published a paper titled “Hierarchically Tiled Array as a High-Level Abstraction for Codelets” in the fourth workshop on Data-flow Execution Models for Extreme Scale Computing (DFM 2014). The paper describes our design of mapping HTA programs to the codelet execution model. The paper can be found here:
http://www.cs.ucy.ac.cy/dfmworkshop/wp-content/uploads/2014/08/DFM2014-10-Hierarchically-Tiled-Array-as-a-High-Level-Abstraction-for-Codelets.pdf

Future Work

- Short Term (Q9)
  - Collaborate with ETI team to clarify the unknown overhead in NAS benchmarks execution for shorter parallel tasks
  - Complete implementation for SPMD mode
- Longer Term (Q10-Q12)
  - 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

During this quarter, we continued to develop our multi-threaded tiling framework to include more techniques, modes of execution and examples. We enhanced jagged tiling to allow applications with concurrent starts to be better represented in our framework. With this implementation we are now able to exploit multi-level concurrent starts. We are currently testing and analyzing the implementation.

We are considering using the Intel Xeon Phi, for both versions of the jagged tiling techniques to strengthen our findings. The current test cases include simple stencil computations (e.g. 3d-7 point stencil), Heat 1d, 2d, 3d, and Jacobi 2d kernel examples. Finally, we refactored our code to simplify the addition and/or modification of features.

Future work

- Q9
  - Add memory re-structuring framework for a third family of applications.
  - Further characterization of other many core designs for jagged tiling and restructuring
- 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:

ETI

For details on the section on "Memory model and semantics" the readers should read the following paper and the citations in the following paper:

Jack B. Dennis and Guang R. Gao: "**On the Feasibility of a Codelet Based Multi-core Operating System" ,** in Proceedings of the 4th Dataflow Models Worshop in conjunction with IEEE PACT-2014, Edmonton, August 24, 2014.

UIUC

Overhead Analysis and Performance Improvements

From the performance results measured in previous quarters, we knew some certain overhead exists in the PIL generated code and the HTA library. The reason is that the performance of HTA-to-OpenMP and HTA-to-SCALE versions are not as good as the pure-OpenMP version of NAS benchmarks. We performed detailed analysis on the NAS benchmarks using performance profiling tools such as callgrind and the ETI swarm tracing library. We discovered the inefficiency in parallel operation invocations were caused by several problems.

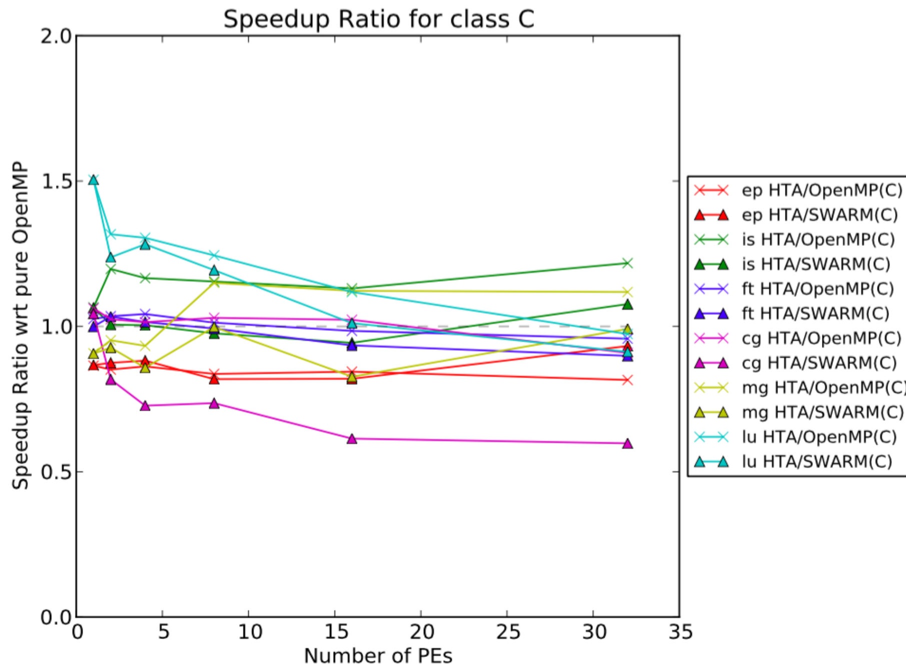
In a discussion with the ETI engineers, we found a potential problem in that the PIL library implement parallel operations using swarm_enter(), which initiates SWARM runtime system each time a PIL operation is invoked through pil_enter() function. The approach induced overhead by initiating and terminating SWARM runtime system for each parallel operation invocation. We changed the PIL compiler to generate code that enter the SWARM runtime system only once in the beginning of the program execution with all subsequent operations being

performed within the runtime system, thus removing the per-invocation runtime system initiating/terminating overheads.

Second, with the help of callgrind, we discovered another source of per-invocation overhead in the allocation/deallocation of the temporary data structures both in PIL generated code and the HTA library. We optimized it by keeping persistent pools of temporary data structures and reusing them whenever possible. Allocations/deallocations only happen when the pool is not large enough. This optimization greatly improves the performance.

With the SWARM tracing library and the trace-viewer tool, we were able to observe the details of program execution. Two unexpected behaviors of the SWARM runtime system were discovered: (1) the SWARM runtime scheduler sometimes schedule work to busy worker threads instead of idle ones, thus creating artificial load imbalance and degrades performance, and (2) for short parallel tasks which are known to have similar workload, some took much longer to complete than the others. We reported the discovery to the ETI team, and they provided an effective solution for the first problem. The root cause analysis of the second problem is on-going.

The following figure shows the ratio of the execution time of the latest HTA-to-OpenMP and HTA-to-SWARM CLASS C versions with respect to the pure OpenMP implementation. The experiment was conducted on a cluster node with 2 Intel Xeon E5-2690 CPUs (total 32 threads). As can be seen in the figure, the HTA versions have performance close to the pure-OpenMP version and sometimes even better. We expect that when the previously mentioned problem (overhead for short parallel tasks) is resolved, the performance results for higher thread counts will further improve.



Reservoir: Scalable Hidden Equalities Detection

Background and problem

Polyhedrons are the base for the representation of programs, data and dependences in the R-Stream automatic parallelization and optimization engine (“the polyhedral mapper”).
There are two dual forms in which polyhedrons can be represented[1]:

- The implicit representation, as the intersection of a set of (linear) constraints. Constraints may be equalities or inequalities.
- The Minkowski representation, as a convex sum of vertices and a non-negative sum of rays.

Duality between both representations entails that the representation of polyhedrons tend to be simple in one form and complex in the other. Intuitively, intersecting a new constraint with a polyhedron typically equates to adding several vertices, and adding a new vertex to a polyhedron typically equates to intersecting the polyhedron with several constraints.
Reservoir’s polyhedral library - named Jolylib - is able to manipulate polyhedrons in either form, and to convert between the forms. However, polyhedral operations such as encountered in program optimization and parallelization are largely simpler in the implicit representation, and converting between forms is computationally expensive. In the course of the DynAX project, Jolylib was hence modified to favor the use of the implicit representation. In order to reduce computations further, the set of constraints representing a polyhedron may be redundant (i.e., not minimal), removal of redundant constraints being done lazily.

Among constraints, equalities play an important role, since they define the smallest linear subspace in which a polyhedron is contained. This defines:

- The geometric dimension of a polyhedron. A polyhedron in a *n*-dimensional space can be reduced to a lower-dimensional polyhedron through (elimination of) its equalities. Also, when analyzing faces of a polyhedron, the face’s geometric dimension tells us if it is a vertex, an edge, or a higher-dimensional face.
- The lattice of integer points spanned by the linear space. Since loop iterations, data and dependences are represented by the integer points of a polyhedron, it is important to know this lattice precisely.

Hence, knowing this subspace for the polyhedrons we manipulate is of great value.
Unfortunately, equalities are not necessarily apparent in polyhedrons. A subset of the polyhedron’s inequalities can indeed define a lower-dimensional subspace, which should instead be represented by equalities. We call these unexpressed equalities “hidden equalities,” and equivalently we say that the subset of inequalities forms a set of equalities (or that they are “equality-forming”).

---

[1] Vincent Loechner, Philippe Clauss, Doran Wilde, Benoît Meister, Rachid Seghir, Gilles Bitran, Julien Léger, “*PolyLib: A Library for doing Symbolic Polyhedra Operations*,” International Symposium on Symbolic and Algebraic Computation (ISSAC’2002), July, 2002.

A naive approach

A simple but naive technique consists of computing the maximum distance between a point in *P* and each inequality of *P*. Inequalities whose distance is zero contribute to a hidden equality. Unfortunately, this costs up to one LP[2] per constraint, which is computationally too expensive in practice.

A scalable approach

The approach we developed this quarter is based on two complementary phases, which recursively classify the inequalities of a polyhedron in three classes: equality-forming, redundant, and non-equality-forming. Note that, since redundant constraints are allowed in the representation, we are lazily detect them, and hence we allow the set of non-equality-forming inequalities to contain (undetected) redundant inequalities.
The first phase classifies a subset of the inequalities of a polyhedron as non-equality-forming. The second phase detects redundant, equality-forming and more non-equality-forming inequalities among the remaining inequalities, and reduces the dimension of the polyhedron. The process continues on the resulting polyhedron, which has lower dimension and typically only a fraction of the non-classified inequalities from the previous polyhedron.
Both steps are presented at a high-level in the next two subsections.

Classifying non-equality-forming inequalities

Equalities of a polyhedron are saturated by all points of the polyhedron (i.e., all points x in P satisfy the equalities Ax+b=0). Hence, if some inequalities are *not* saturated by a particular point of the polyhedron, then they do not contribute to an equality. The cone-forming part of each step uses LP to find an extremal point of the currently considered polyhedron S, and finds the set C of inequalities saturated by this point. Inequalities not saturated by the point are classified as non-equality-forming. Note that this step typically reduces drastically the number of constraints left to be classified. Because S is convex and C are all saturated by a common point, C forms a cone, which is further classified by the next step.

Classifying more inequalities and reducing the dimensionality of the problem

By analyzing the possible combinations of the inequalities incident to C, the algorithm detects:

- redundant inequalities, which are a positive linear combination of constraints of C,
- equality-forming inequalities, which are a negative linear combination of constraints of C, and
- non-equality-forming, non-redundant inequalities, which are independent of all the other inequalites of C.

After classifying these inequalities, a lower-dimensional polyhedron P’ is computed by finding a hyperplane that intersects the half-spaces represented by all the remaining inequalities of C, and intersecting C with the hyperplane.

[2] i.e., the solving of a Linear Programming problem.

The next step, made of the two above mentioned phases, is repeated on the simpler, lower-dimensional polyhedron P'.

Complexity

The algorithm reduces the dimension of the problem by at least one at each step (more if hidden equalities are detected during the step). The cost of each step is dominated by the solving of a single LP, which is polynomial in the number of dimensions of P. So one theoretical bound to the computational complexity of this algorithm is polynomial in the number of dimensions of P, the polynomial being one degree higher than the cost of LP.

Empirically, it turns out that this algorithm is much faster than the naive algorithm presented above, even though they have comparable complexities. Its complexity is definitely practical for use in a polyhedral compiler, as opposed to the naive solution. Good insight into the practical complexity of our algorithm is obtained by trying to build an adversarial case. Polyhedrons with a high number of constraints quickly see these reduced in Phase 1. On the other hand, redundant and equality-forming inequalities are more likely to be found in polyhedrons with a low number of constraints (irrespective of the dimension). One could argue that Phase 1 doesn't classify any constraint whenever P is already a cone. However, in this case simplex-based LP algorithms also tend to return faster.

Status

The algorithm was implemented, tested and incorporated in the current development version of Jolylib. It will be part of the next release of R-Stream, slated for September.
A complete description of the algorithm, along with all the necessary proofs, is available on demand.

PNNL

The concept behind group locality and jagged tiling are simple, yet powerful optimization techniques. At the current iteration, we take advantage of the polyhedral formulation to calculate dependencies, hyperplanes and schedules. The jagged tiling framework uses this information to modify the incoming iteration space based on the available parallelism and the locality that could be exploited from different levels in the hierarchy.

When using current polyhedral framework, the tiling techniques determinate shapes and composition based on the minimization of communication between the tiles, but they do not consider the differences between the levels of the memory hierarchies or the parallelism that might be left on the table. Jagged tiling, using the techniques presented in the LCPC 2014 submission, tries to take advantage of the parallelism and locality for stencil applications.

In here, we present a sample set of experiments that were done on Intel Xeon Phi 7110P coprocessor. Each coprocessor is equipped with 61 cores running at 1.1 GHz and supports up to 4 hyper-threads. Figure 1 and Figure 2 show the performance of eSidel-1d and Seidel-2d

stencil application compared against PLUTO generated OMP code. In addition, Figure 3 and Figure 4 show memory reads and writes for Sidel-2d experiments acquired using performance counters. All experiments were run using 4 threads per group such that all threads within a core form a thread group and take advantage of the shared structures of the architectures (i.e. the Level 2 cache in the case of the Intel Phi). Under this configuration, we can take advantage of the L2 cache locality while enhancing the performance of the L1 tiles by using a fine grain execution model. Our results show that using jagged tiling along with thread grouping we are able to reduce number of access to the memory (shown in figures 3 and 4 for the 2D case), hence improving performance (shown in figures 1 and 2 for 1D and 2D cases).

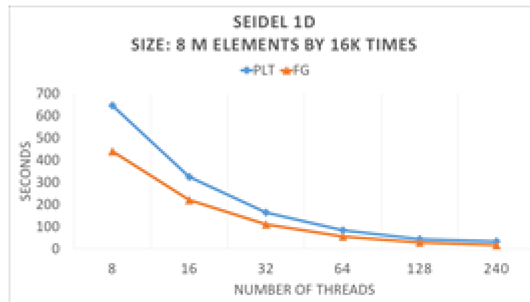


Figure 1: Seidel 1D kernel performance - PLT is PLUTO generated code and FG is the Jagged Tiled version

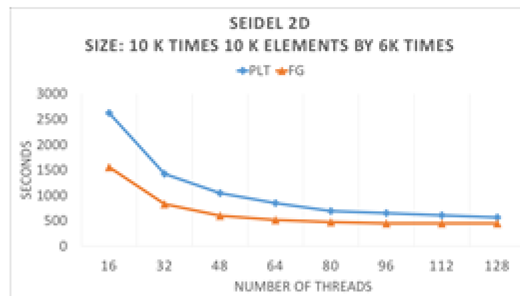


Figure 2: Seidel 2D Kernel Performance

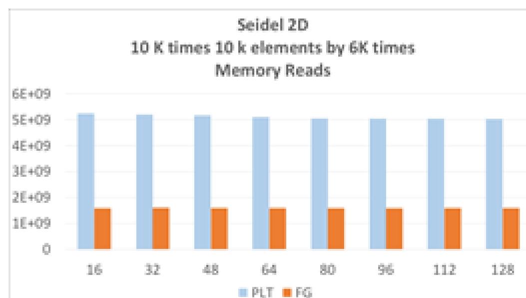


Figure 3: Seidel 2D Memory Reads serviced by the Main Memory

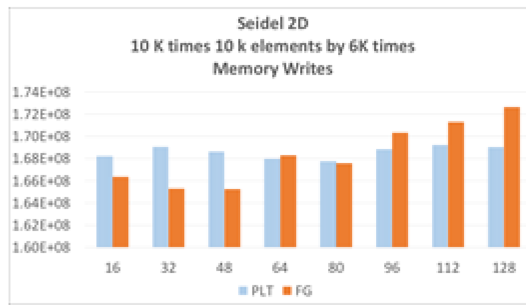


Figure 4: Seidel 2D Memory Writes serviced by the Main Memory