High-level Status Summary

Technology	Description (Institution)	Status
Applications/Runtime	TCE mapping/porting to OCR block level (ETI)	Done
Memory access semantics/Runtime	TCE mapping/porting to SWARM block level (ETI)	In Progress
Resilience	Containment Domains (ETI)	Evaluating
Application migration	MPI Interoperability (ETI)	Evaluating
Parallelizing compiler	Support for distributed data and computation placement	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)	Done
Applications	Lulesh refactoring (PNNL)	Done
Enhanced Data Types	Design of Composite Data Types (PNNL)	In Progress

Summaries of Quarterly Work (Q9)

ETI Work

During this reporting period (Q9: 09/01/2014-12/31/2014), 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 8.1: Research integration of containment domain execution and recovery with codelet scheduling (in progress)

Task 10.1: Study MPI interoperability (in progress)

During this quarter, main progress was made on TCE 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.

Progress also made on Task 8.1 (resilience) and Task 10.1 (MPI Interoperatability) as described below.

We have also worked in collaboration with UIUC, Reservoir and PNNL to organize our results and presenting them in poster form at SC2014. Much interactions between the team members have had happened during this process.

A new SWARM release is posted for our partners under this project as well as elsewhere to access the latest results in this front.

TCE

To facilitate the readers to understand the progress of our TCE task, the readers may find some historical background useful. This background detail is included in the “Topic Detail” section, although it first appeared in an earlier report (Q6).

ETI has continued to adapt the TCE code generator to generate task-based runtime code. In the previous quarters, we got basic code generation working for the Open-Community Runtime (OCR) and the SWARM runtime. This preliminary adaptation gave 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 successfully finished adapting the TCE proxy application to the OCR runtime with full block-level parallelism, and have begun implementing a similar block-level adaptation for the SWARM runtime.

This is a process of decomposing the problem further, to take advantage of the block-sparse tensor data format in use. This allows 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 level of parallelism is necessary to produce a scalable version of the application, as TCE applications contain few tensors, but each tensor contains many blocks. CCSD is an example of a typical TCE application, which generates around 40 tensors during the course of execution, with a dependency graph constraining which tensors can be generated at any given moment. As the problem size grows, this number of tensors is fixed, but the number of blocks per tensor increases exponentially. A scalable implementation of TCE must be able to take advantage of the increasing number of blocks, and expose a corresponding increase in the amount of parallelism.

The block-parallel implementation 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 distributed across compute nodes, how to organize the computation and the other data in order to minimize the necessary communication between compute nodes, and how to ensure that no single compute node overflows its local memory during the course of execution.

The OCR generator exposes the full amount of parallelism and demonstrates correct operation with a task-parallel execution model, but it does not attempt to tackle the challenges of task prioritization, data locality, or memory management. At time of writing, we believe that the SWARM runtime has better tools for addressing these issues.

This work has 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. In both instances, 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.

Resilience (containment domains)

This quarter, we have begun our research into resilience using containment domains. Through conversations with Mattan Erez and his team at UT Austin, we have developed a plan for pursuing our research.

We have created a prototype implementation of containment domains in SWARM. This implementation allows a user to create a containment domain for a codelet complex, check for errors when it is complete, and re-run the codelet complex if errors are detected. Any necessary inputs can be preserved on the first run, and copied back on subsequent runs. The current implementation has a few limitations, which are currently being fleshed out.

Once this is done, we will start looking at how to apply containment domains to real applications. We will investigate the trade-offs between various arrangements of containment domains in an application, and compare them to standard methods of resilience such as checkpointing.

MPI Interoperability

The DynAX project needs to interoperate with legacy MPI codes. Because the codes are being modified and recompiled to fit into a new exascale paradigm, we assume that the codes can be recompiled through the X-Stack software. We also note that interoperability with MPI should not degrade the current performance of legacy codes, but it may hinder optimal performance and programmer intervention may be required to remove bottlenecks.

Legacy code can be parallelized between MPI calls rather straightforwardly. The main MPI thread will be suspended while the parallel code is executed, then the main thread is resumed in a manner similar to how OpenMP and MPI interoperate today. However, this method is limited because only the main MPI thread may make MPI calls. We will study ways of allowing MPI calls to be made in codelets, possibly by wrapping blocking MPI calls in non-blocking functions with codelet semantics.

DynAX is not the only X-Stack project which is interested in interoperability between task-parallel execution models and legacy execution models. In particular, the XPRESS team lists application migration as one of their objectives, too. We believe there is likely to be some overlap between the DynAX goals and the XPRESS goals. In this quarter, we have begun to reach out to the XPRESS team, and have had several interactions with their chief scientist on the subject.

SWARM

A new version of SWARM was released on November 4, version 0.16. This release improves the caller/callee protocol on distributed memory machines, and addresses several performance issues.

Reservoir Work

Distributed data and computation placement

This quarter, we have implemented initial support for automatic parallelization to clusters in R-Stream. This is a necessary enabling step for the following SOW tasks:

Task 2.4: Research compiler code generation for data placement and movement
Task 3.4: Optimize unstructured computations

The unstructured codes we are considering are all run on a distributed environment, and data placement is a key optimization aspect of these codes. Data placement is intrinsically related to computation placement, and a key question in data placement is whether it is better to migrate data or computations. Hence, we are developing a framework that enables dynamic data and computation placement.

Our approach, explained in the “Distributed data and computation placement” section below, departs from the naive belief that the best way of targeting a hierarchy of memories is to tile the code (and possibly the data) for every level of memory.

R-Stream

We plan to release a snapshot of R-Stream 3.3.4 in the first week of December. The snapshot will feature scalable codelet dependence generation, and a more robust raising phase, which integrates modulo conditionals in the iteration domain. Modulo conditionals are found for instance in red-black solvers.

Future work

In Q10, we plan to complete the implementation of our cluster backend. In Q11-Q12, we plan to adapt this backend to deal with unstructured or block-structured mesh computations. We will also keep working on incremental improvements along all the aspects of the project.

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)

SPMD Execution Mode

We extended the PIL compiler to accept SPMD PIL programs. Since processes can synchronize with each other through point-to-point synchronization primitives, programs written in SPMD style do not need global barriers which are typically assumed for data parallel (array) operations. Avoiding global barriers can greatly improve application performance.

We finished implementing the OpenMP backend of the SPMD PIL compiler, and updated the SPMD PIL programming interface and point-to-point communication primitives. This allows us to start implementing HTA library and efficient collective communication primitives using SPMD PIL. We have also started implementing SCALE backend (running SWARM 0.16) for the SPMD PIL compiler and we expect to complete it in the next quarter. The HTA library implementation of SPMD mode will also be finished in Q10 along with any modifications required for NAS Parallel Benchmarks to run in SPMD.

SPMD execution also enables the HTA library to deal with irregular parallelism. We developed a strategy using the SPMD mode to execute tiled LU factorization algorithm. Please see the Details section for more information.

Future Work

- Short Term (Q10)
  - Complete implementation for SPMD mode
  - Evaluate NAS Parallel Benchmarks performance in SPMD mode
  - Implement tiled LU factorization algorithm to benchmark SPMD mode performance
- Longer Term (Q11-Q12)
  - Implement a variety of other benchmarks
  - Evaluate and tune for performance
  - Evaluate programmability using objective metrics (e.g. number of operations)

PNNL Work

Research Status

Group Locality (GL)

During this quarter, we concluded several experiments on our multi-threaded tiling framework and its new techniques (like Jagged Polygon Tiling). The Jagged Polygon Tiling technique allows hierarchical concurrent start for memory hierarchy aware tile groups. As in the previous research iteration of Jagged Tiling, each execution schedule and tile shape exploits the available parallelism, load balance and locality present in the given applications. Our base architecture is the Intel Xeon Phi architecture with selected

and representative stencil kernels. We show improvement ranging from 5.58% to 31.17% over existing state-of-the-art techniques.

Architected Composite Data Types (ACDT)

We are in the process of porting our ACDT framework to the distributed SWARM framework. We expect to see even larger performance and power efficiency gains for selected kernels over the single node version, as we leverage larger latencies across nodes.

Future directions

- (GL) Add memory restructuring framework for a third family of applications.

- (GL) Further characterization of other many core designs for Jagged Polygon Tiling and restructuring

- (ACDT) Distributed SWARM experiments

Publications

- (ACDT) Accepted: ICPADS 2014

- (GL) Accepted: CGO 2015

- (GL) Accepted: SC14 Poster, Best Poster Nominee

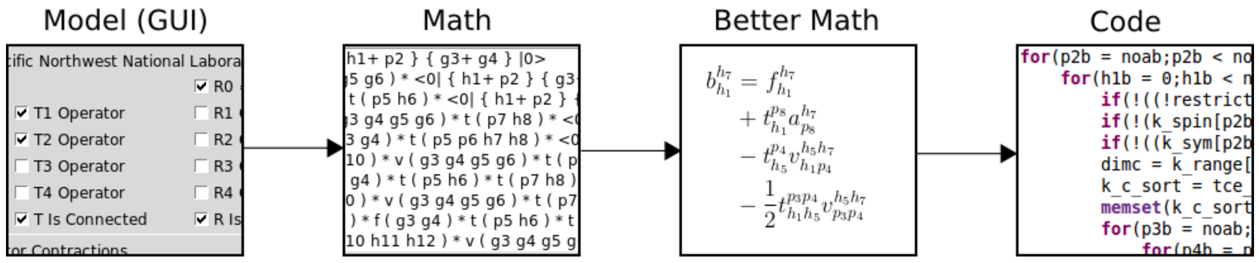
Topic Detail:

ETI

TCE, A proxy app for Shrödinger equations in NWChem

Introduction

The Tensor Contraction Engine (TCE) is a feature of NWChem, a computational chemistry package maintained by PNNL[1]. It is used by NWChem to solve electrical Shrödinger equations.



```
h1+ p2 } { g3+ g4 } |0>
5 g6 ) * <0| { h1+ p2 } { g3
t ( p5 h6 ) * <0| { h1+ p2 }
3 g4 g5 g6 ) * t ( p7 h8 ) * <
3 g4 ) * t ( p5 p6 h7 h8 ) * <0
10 ) * v ( g3 g4 g5 g6 ) * t ( p
g4 ) * t ( p5 h6 ) * t ( p7 h8 )
0 ) * v ( g3 g4 g5 g6 ) * t ( p7
) * f ( g3 g4 ) * t ( p5 h6 ) * t
10 h11 h12 ) * v ( g3 g4 g5 g
```

$$b_{h_1}^{h_7} = f_{h_1}^{h_7} + t_{h_1}^{p_8} a_{p_8}^{h_7} - t_{h_5}^{p_4} v_{h_1 p_4}^{h_5 h_7} - \frac{1}{2} t_{h_1 h_5}^{p_3 p_4} v_{p_3 p_4}^{h_5 h_7}$$

```
for(p2b = noab;p2b < no
    for(h1b = 0;h1b < n
        if(!((!restrict
        if(!(k_spin[p2b
        if(!((k_sym[p2b
        dimc = k_range[
        k_c_sort = tce_
        memset(k_c_sort
        for(p3b = noab;
```

The core of TCE is a Python library. The TCE library includes a GUI to input the details of a particular correlation model, produces tensor contraction expressions to implement that model, performs various optimizations on those expressions at the abstract level (such as factoring and reuse of common sub-expressions), and generates Fortran code. The above

[1]For more information on NWChem, see their website (nwchem-sw.org).

diagram gives a high level visualization of this process. The generated Fortran code is run iteratively by NWChem to calculate various properties of the input molecule, using the correlation model selected by the input file. There are many correlation models[2], so there is quite a lot of code in NWChem which was generated by this script, nearly 3 million lines of code in more than ten thousand files.

TCE is a large and interesting component of NWChem. However, the fact that it is embedded into NWChem makes it difficult to work on directly. For meaningful data sets, it may take NWChem several hours of CPU time to generate the input data necessary to run the TCE step. Additionally, the use of Fortran is a compatibility problem for the runtimes and tools we are working on in the DynAX X-Stack project. Therefore, we saw a benefit to extracting it into a standalone application which can be run directly, implemented in a programming language that is supported by the SWARM and OCR task-based runtimes.

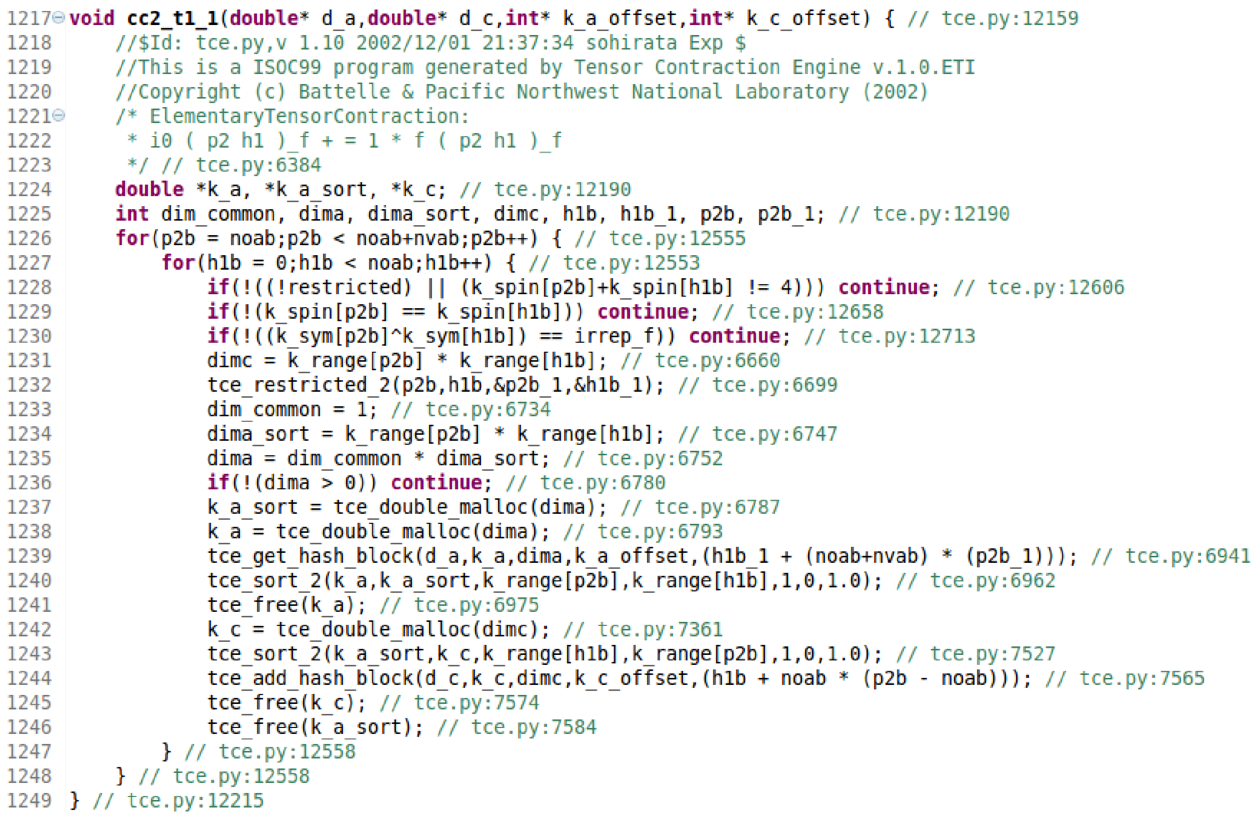
Proxy App

We've taken TCE and turned it into a standalone proxy app for doing research. We have adapted the Python libraries to generate serial C code, and provided a minimal set of library functions necessary to run this code directly. We have also produced several data sets, consisting of all of the inputs and parameters that NWChem runs it with for a single iteration. These data sets also include reference output data from NWChem, which is compared with freshly generated data to ensure accuracy. The C code is compiled into a standalone program, which is run with the input file names specified on the command line, runs the set of tensor contraction expressions, generates the output tensor, compares that to the reference output data from NWChem, and reports the validity of the results.

This work has focused almost exclusively on the final step of the Python libraries, namely, the code generation step. Using the Fortran generation code as a reference, we implemented a separate code path which emits serial C code. The resulting library can emit either Fortran or C, depending on which method you call. The structure of the generated Fortran and C code is as similar as possible, apart from being serial. (The parallel API used in NWChem is very simple, so the difference of making the code serial is very small.) This similarity is important, as it may allow us to keep Fortran in sync as we analyze and make improvements on the C side.

A simple generated C function may look like this:

[2] Link to TCE correlation models (nwchem-sw.org)



```c
void cc2_t1_1(double* d_a,double* d_c,int* k_a_offset,int* k_c_offset) { // tce.py:12159
    //$Id: tce.py,v 1.10 2002/12/01 21:37:34 sohirata Exp $
    //This is a ISOC99 program generated by Tensor Contraction Engine v.1.0.ETI
    //Copyright (c) Battelle & Pacific Northwest National Laboratory (2002)
    /* ElementaryTensorContraction:
     * i0 ( p2 h1 )_f + = 1 * f ( p2 h1 )_f
     */ // tce.py:6384
    double *k_a, *k_a_sort, *k_c; // tce.py:12190
    int dim_common, dima, dima_sort, dimc, h1b, h1b_1, p2b, p2b_1; // tce.py:12190
    for(p2b = noab;p2b < noab+nvab;p2b++) { // tce.py:12555
        for(h1b = 0;h1b < noab;h1b++) { // tce.py:12553
            if(!((!restricted) || (k_spin[p2b]+k_spin[h1b] != 4))) continue; // tce.py:12606
            if(!(k_spin[p2b] == k_spin[h1b])) continue; // tce.py:12658
            if(!((k_sym[p2b]^k_sym[h1b]) == irrep_f)) continue; // tce.py:12713
            dimc = k_range[p2b] * k_range[h1b]; // tce.py:6660
            tce_restricted_2(p2b,h1b,&p2b_1,&h1b_1); // tce.py:6699
            dim_common = 1; // tce.py:6734
            dima_sort = k_range[p2b] * k_range[h1b]; // tce.py:6747
            dima = dim_common * dima_sort; // tce.py:6752
            if(!(dima > 0)) continue; // tce.py:6780
            k_a_sort = tce_double_malloc(dima); // tce.py:6787
            k_a = tce_double_malloc(dima); // tce.py:6793
            tce_get_hash_block(d_a,k_a,dima,k_a_offset,(h1b_1 + (noab+nvab) * (p2b_1))); // tce.py:6941
            tce_sort_2(k_a,k_a_sort,k_range[p2b],k_range[h1b],1,0,1.0); // tce.py:6962
            tce_free(k_a); // tce.py:6975
            k_c = tce_double_malloc(dimc); // tce.py:7361
            tce_sort_2(k_a_sort,k_c,k_range[h1b],k_range[p2b],1,0,1.0); // tce.py:7527
            tce_add_hash_block(d_c,k_c,dimc,k_c_offset,(h1b + noab * (p2b - noab))); // tce.py:7565
            tce_free(k_c); // tce.py:7574
            tce_free(k_a_sort); // tce.py:7584
        } // tce.py:12558
    } // tce.py:12558
} // tce.py:12215
```

The code operates on tensors in a block-sparse data format, taking advantage of several kinds of symmetry to compact the data and improve locality. As a result, the data structures are fairly complex when you first see them. ETI gave a technical deep dive on TCE in December, in which we described the details of the data structures, as well as the actual math, the function APIs, details of inputs and outputs, and where the code sits in the context of the overall NWChem application. If there is an interest in these details, the slides can be found here[3].

Availability

As mentioned above in the status section, ETI has been working to adapt the proxy app to run on task-based runtimes, and analyze its performance. Since this work may be useful for other research teams, it is being made available for all to use. Several versions of the code, corresponding to major milestones in the adaptation process, can be found on the DynAX website[4].

[3] Link to DynAX TCE deep dive slide deck (xstackwiki.com)
[4] DynAX website (xstackwiki.com)

UIUC

Tiled LU Factorization in SPMD

Tiled LU factorization algorithm can be expressed in HTA notation in a few lines of code:

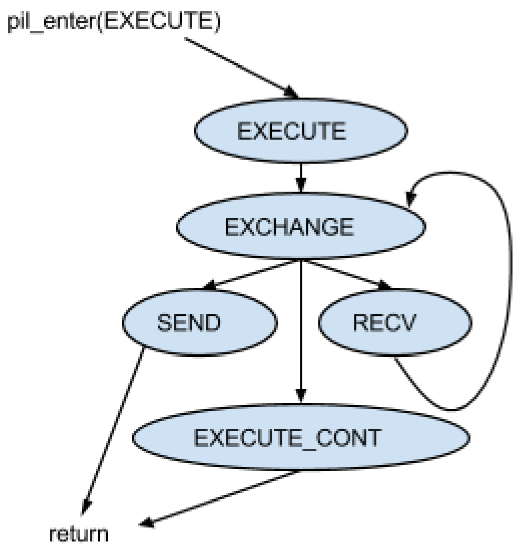
```
for k = 0 to (n-1) {
    lu (A(k, k));                                                       // Local
    A(k, k+1: ) = mldivide(A(k,k).lt, A(k, k+1: ));                     // Row k
    A(k+1: , k) = mrdivide(A(k+1: , k), A(k, k).ut);                    // Column k
    A(k+1: , k+1: ) = A(k+1: , k+1: ) - A(k+1: , k) * A(k, k+1: );     // Submatrix
}
```

In practice, the HTA operations can be expanded into parallel loops with each iteration working on several tiles. There are no dependences across iteration of the parallel loop, but in a naive implementation the different loop iterations must be executed sequentially. However, with the SPMD implementation, the iteration space of parallel loops can be partitioned and executed in different ``processes'', processes synchronize using point-to-point communication mechanisms instead of global barriers. Please refer to our paper published and reported last quarter about the mechanism to execute SPMD programs in codlet runtime.

Chih-Chieh Yang, Juan C. Pichel, Adam R. Smith, David A. Padua. *Hierarchically Tiled Array as a High-Level Abstraction for Codelets.* In the Fourth Workshop on Data-Flow Execution Models for Extreme Scale Computing, 2014

Each process, P, executes the following code in SPMD fashion. The EXECUTE() function examines the input, output and owned tiles in order to (1) determine the communication required with other processes (when for some process Q, some of the input tiles are owned by P but the output tile is owned by Q), (2) perform the computation (when P owns the tile on the left hand side of the assignment), or (3) skip the computation if P does not own any input nor output tiles of the EXECUTE().

```
  for k = 0 to (n-1) {
    EXECUTE(lu, {A(k, k)}, {A(k, k)});                            // Local
    for j = k+1 to n-1 {
        EXECUTE(mldivide, {A(k, j), A(k,k).lt}, {A(k, j)});       // Row k
    }
    for i = k+1 to n-1 {
        EXECUTE(mrdivide, {A(i, k), A(k, k).ut}, {A(i, k)});      // Column k
    }
    for j = k+1 to n-1 {
        for i = k+1 to n-1 {
            EXECUTE(dgemm, {A(i, j), A(i , k), A(k, j)}, {A(i, j)}, 1.0, -1.0);
        }
    }
  }
```



EXECUTE implemented in PIL

Reservoir

Distributed data and computation placement

A prerequisite for studying data placement in deep hierarchies and realistic unstructured mesh codes is to support distributed memory systems (clusters). A key aspect of these systems that is expected to be present in Exascale systems is a long-latency, low-bandwidth link among the portions of the coarsest levels of the memory hierarchy. One example of this is the Intel Traleika Glacier architecture[5], which implements bandwidth tapering.

We have started implementing support for clusters in R-Stream, with the following design principles.

No hierarchical tile spawning. We learned that, for non-embarrassingly-parallel codes, the hierarchical spawning of tasks through hierarchical tiling introduces artificial collective dependences[6] and, if applied naively, reproduces non-parallel startup and finish issues at every level of the task hierarchy. Recent work by PNNL within the DynAX program showed a method in which decoupling different levels of tiling prevents the non-parallel startup problem from happening at the second level of tiling[7]. However, artificial coarse-grain synchronizations

[5] Carter et al. "Runnemede: An architecture for Ubiquitous High-Performance Computing." In Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture, HPCA 2013.

[6] Hierarchical decompositions were obtained in our previous experiments through a recursive decomposition of codes, based on parametric tiling. In Vasilache et al., "A Tale of Three Runtimes", research report from Reservoir Labs. http://arxiv.org/abs/1409.1914. We addressed the hierachical result and discussion in more depth during a presentation at the 2014 SIAM Conference on Parallel Processing for Scientific Computing. https://live.blueskybroadcast.com/bsb/client/CL_DEFAULT.asp?Client=975312&amp;PCAT=7954&amp;CAT=7955

[7] Shresta et al, "*Jagged Tiling for Intra-tile Parallelism and Fine-Grain Multithreading*", in proceeding of The International Workshop on Languages and Compilers for Parallel Computing, LCPC 2014, Hillsboro, OR.

are still introduced, resulting in non-optimal efficiency. In order to avoid this problem, we will place tasks working on distributed memories and compute nodes without decomposing the code into hierarchically spawned tasks (which is usually obtained through hierarchical tiling).

Decoupling data reuse and parallelism. There is a well-established trade-off between temporal locality and parallelism, which, within individual codelets, is addressed by R-Stream through its static scheduler. In the context of codelets dynamically scheduled and operating on deep hierarchies of memory, the trade-off is as follows (illustrated on Figure 1):

- Codelets can work by fetching data from a far memory F to a close one C.
- When there is an intermediary memory M, and when there is enough data reuse among a set T of codelets that have access to same piece of F, it is profitable to create a copy of the incoming shared data into M, because transferring the data directly from F to C results in redundant data transfers. The codelets then bring in their input data from a data set (let us call it S) located in M, and data locality is improved.
- Unfortunately, this implies that the whole set T of codelets that receive their data from S cannot start working until S is formed. This translates into a collective synchronization between the filling of S to the codelets of T.
- Similarly, the codelet(s) that update S must wait for S to be up-to-date (for T) before the copy. In other words, a collective synchronization is necessary between the preceding writes to S's data and the fill-in of S.
- The reasoning is similar with outgoing data.

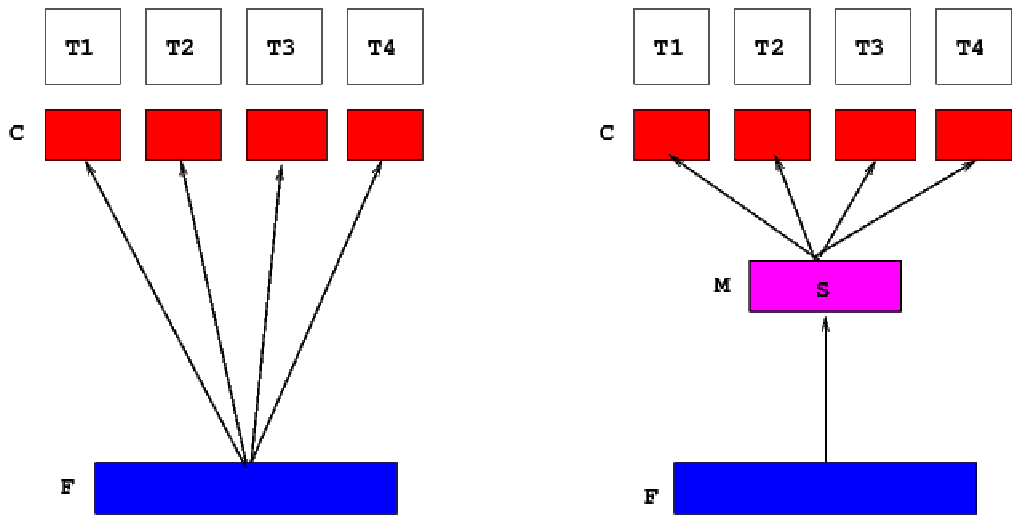


Figure 1. Locality-Parallelism trade-off in the use of intermediate data.

In sum, the use of intermediary data generally results in additional collective synchronizations, i.e., less parallelism. A number of decisions will impact the performance of the mapped application in this respect, including:

- Whether to copy data to an intermediary data structure. It is clearly only profitable when enough reuse is happening among the codelets that access the data.
- We have also developed strategies to alleviate the need for collective synchronizations.

Decoupling data and computation migration granularities. In a system with billions of codelets spread across different nodes, scheduling (migrating) tasks individually across nodes does not seem scalable. We plan to support group migration to address this issue. We will also permit the definition of data tiles whose sizes are independent of the size of tasks (or groups of tasks).

Non-SPMD execution model. Last but not least, SPMD parallelization results in a lack of load balancing. We will then use the same high-level scheduling principles across nodes as the ones SWARM applies within nodes to avoid this issue.

We have assessed that Global Arrays (GAs) will allow us to reason with tiled distributed data and explicit data movement to a large extent. We have implemented a data movement runtime based on GAs, and a test backend for it. The test backend, which produces distributed SPMD codes, allowed us to fix bugs in our runtime layer implementation. The generated test codes run one thread per node rank.
With this backend as a starting point, we are moving towards fulfilling the other features presented herein.

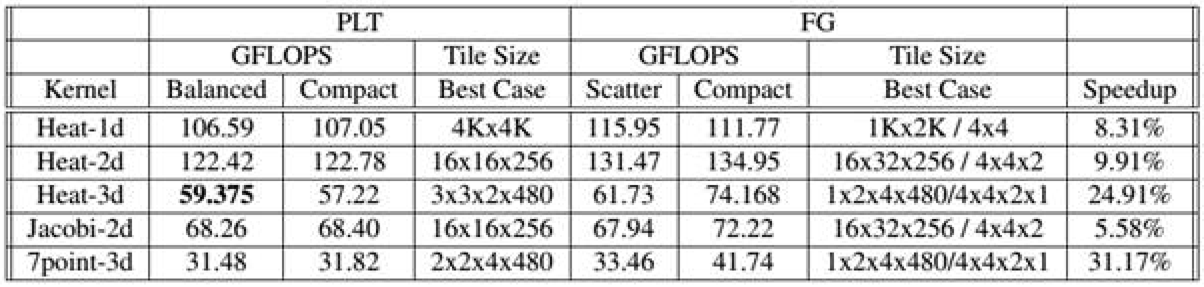
PNNL

Jagged Polygon Tiling

The concepts behind Group Locality and Jagged Tiling are simple, yet powerful optimization techniques. At the current research iteration, we take advantage of the polyhedral formulation to calculate dependencies, hyperplanes and schedules. The Jagged Polygon 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.

Current, existing polyhedral framework techniques determine tile shapes and composition based on the minimization of communication between tiles, but they do not consider differences between levels of the memory hierarchies or the parallelism that might be left on the table. As in Jagged Tiling, we take advantage of the parallelism and locality for stencil applications - we introduce the Jagged Polygon technique in the CGO 2015 paper.

Figure 2 presents a case study of state-of-the-art PLUTO generated code versus the code generated by the Jagged Polygon Tilling. The advantage over the PLUTO code ranges from around 5% to 25%. An interesting observation is that a higher stencil dimensionality favors the Jagged Polygon framework. This can be explained by the reduction of locality (due to striding) in higher dimensions.



	PLT			FG			
	GFLOPS		Tile Size	GFLOPS		Tile Size	
Kernel	Balanced	Compact	Best Case	Scatter	Compact	Best Case	Speedup
Heat-1d	106.59	107.05	4Kx4K	115.95	111.77	1Kx2K / 4x4	8.31%
Heat-2d	122.42	122.78	16x16x256	131.47	134.95	16x32x256 / 4x4x2	9.91%
Heat-3d	**59.375**	57.22	3x3x2x480	61.73	74.168	1x2x4x480/4x4x2x1	24.91%
Jacobi-2d	68.26	68.40	16x16x256	67.94	72.22	16x32x256 / 4x4x2	5.58%
7point-3d	31.48	31.82	2x2x4x480	33.46	41.74	1x2x4x480/4x4x2x1	31.17%

Figure 2: Performance of State of the Art and Jagged Polygon execution for selected kernels