High-level Status Summary

Technology	Description (Institution)	Status
Applications/Runtime	TCE modification and mapping/porting to OCR (ETI)	Done
Applications/Runtime	TCE mapping/porting to SWARM (ETI)	In Process
Memory access semantics/Runtime	Evaluation of memory semantics based on TCE mapping to OCR and SWARM (ETI)	Evaluating
Data placement topologies/Runtime	Report on the progress made based on TCE mapping and porting (ETI)	Done
Parallelizing Compiler	Exascale-friendly codelet code generation scheme (Reservoir)	Done
Parallelizing Compiler	Integration of the x86 virtual DMA API in the SWARM R-Stream target (Reservoir).	Evaluating
Parallelizing Compiler	Bulk communication abstraction layer (Reservoir).	In Process
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
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 (Q6)

ETI Work

During this reporting period (Q6: 12/01/2013-02/28/2014), ETI has been working on the following tasks, according to the SOW.

Task 2.1: Research data placement topologies (in progress)

Task 2.2: Research memory access semantics (in progress)

Task 7.1: Define intermediate representation (in progress)

During this quarter, main progress was made in connection with Task 2.1 and Task 2.2. 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 2.1 and 2.2 -- 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. We also briefly studied LULESH as well as other benchmarks in the DoE domain. We are in the process of planning a publication[1] on the preliminary results of tasks 2.1-2.3.

TCE

ETI has continued to work on the simple C test program we started work on last quarter. We have extended it to work with NWChem's "2eorb" storage format for two-electron integrals. This significantly reduces the memory footprint, allowing it to run larger problems. This storage format extension has allowed us to successfully run an iteration of CCSD on a Benzene molecule. This operation takes 18 seconds to perform. This should provide enough system load to start doing scalability analysis, though we expect somewhat larger data sets (for correspondingly larger molecules) to work as well.

ETI has begun to adapt the code generator to generate task-based runtime code. The Python code generator has been extended to allow it to generate Open-Community Runtime (OCR) code. The generated OCR code uses one OCR Event-Driven Task (EDT) per tensor contraction expression, and one OCR data block per tensor. Sibling tensor contractions may run in parallel, and their outputs will be summed together to produce the full value of a tensor.

[1] Toward a collection of proxy apps for DynAX studies

Additional "sum" EDTs are added for this purpose. We are currently working on generating SWARM code which operates in a similar fashion.

This task-based runtime code gives us a very coarse level of parallelism. The parallelism is limited to the number of tensor contraction EDTs, and those EDTs vary wildly in execution time, resulting in very poor load balance, and very poor utilization of processing resources. The next step will be to decompose the problem further, to have one data block per block of tensor data, and one EDT for every block-level calculation. This will parallelize the critically large (order N^5) tensor contraction expressions, and allow us to start studying the data placement and data movement characteristics of this application.

Reservoir Work

During the Q6 period, Reservoir has contributed to the following SOW tasks:

Task 2.4 - Compiler code generation for data placement and movement
Task 3.2 - SWARM code generation (tuned)
Task 5.8 - PIL-to-R-Stream mapping path

Exascale-friendly code generation scheme

The R-Stream backend and runtime, based on "autodec" synchronization and spawning operations, are mature enough to be included in the forthcoming release of R-Stream (version 3.3.3).

Autodecs are implemented on top of SWARM's "counted dependence" synchronization mechanism. A counted dependence associates a codelet with a counter, initialized to a positive number representing the number of codelets it depends on. The value of the counter is decremented when a predecessor codelet completes, and when it reaches zero the associated codelet is scheduled.

When finished, a codelet issues “autodecs”, which decrement the counters associated with their successor tasks in the task dependence graph. If the counted dependence is absent, the codelet atomically initializes it and decrements it by 1 automatically.

A paper on autodecs has been submitted to the International Conference on Supercomputing. The paper compares the autodec synchronization mechanism to synchronization mechanisms available in Exascale-related runtimes SWARM, Open Community Runtime (OCR) and Intel's Concurrent Collections. The comparison is made in terms of the overheads of each environment when programming to them, as opposed to the overheads associated with their individual operations.

x86 virtual DMA

In the last report we described an API to perform non-temporal transfers between DRAM and caches on x86 machines, and their intended use in support of R-Stream virtual scratchpad technology. Since the API is that of a two-dimensional Direct Memory Access engine, we called it a "virtual DMA" API.

This quarter, we modified the R-Stream SWARM backend to enable the use of (virtual) DMA transfers, and we created a modified machine model reflecting the fact that data transfers between DRAM and cache are performed explicitly through the virtual DMA engine. Preliminary results are presented in Section "x86 virtual DMA experiments."

Bulk communication abstraction layer

The success of virtual DMAs is encouraging us to investigate the extension of DMA transfer into abstractions to target irregular codes, such as blocked sparse codes. We are also investigating how Architected Composite Data Types, developed by the PNNL team, could contribute to this effort. This work contributes towards Task 3.4 of the SOW.

PIL-to-R-Stream mapping path

We worked on supporting UIUC in their effort to target R-Stream. A first successful experiment was run on matrix multiply after we fixed a bug in the R-Stream front-end, in which a PIL type name was not maintained all the way through parallelization.

UIUC Work

Task 4.1: Representation of sparse arrays (completed in Y1)

Task 4.2: Irregular tiles (in progress)

Task 5.1’: Design of the PIL API (completed in Y1, extension expected to complete in Y2 Q3)

Task 5.2’: Implementation of the PIL API (in progress)

Task 5.3’: Evaluation of the PIL implementation and API (will start in Y2 Q3)

In this quarter, the UIUC team designed a strategy for mapping of application programs implemented in the SPMD programming model onto distributed memory SCALE (Task 5.1’ & 5.2’). Software developers can program in HTA, which is implemented in the SPMD programming model provided by PIL. PIL compiler maps the SPMD code into SCALE’s codelets to let the underlying SWARM runtime system automatically discover and exploit the parallelism hidden, schedule execution dynamically, and move data according to runtime information or compiler hints. We finished the HTA-to-PIL interface design for SPMD in this quarter (a shared memory version has been operational for two quarters), and we expect to have a working implementation in the next quarter. The support for irregular tiles (Task 4.2) will be delayed until the next quarter due to the significant changes in both the HTA library and PIL compiler to support the SPMD programming model.

PNNL Work

During this quarter, we finished the Lulesh refactor and created two methodologies aimed to increase performance and power efficiency.

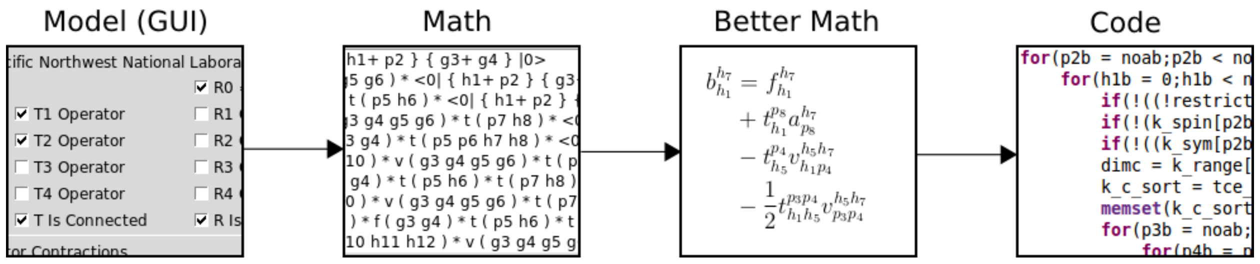
With respect to the refactor, we moved the code from C to C++, switched from a global data structure, to local data structures, and condensed all functions into 4 CnC steps. We created two methodologies for very fine-grained execution models, such as SWARM. The first method is an incarnation of the *Rescinded Primitive Data Types* (RPDT) paradigm, called *Architected Composite Data Types* (ACDT) that aims to provide a runtime-aware system with hints from users (e.g., compilers, system designers). It can transform data layouts, use more efficient operators, and use more efficient access pattern according to application / architecture needs or state (T9.1). The second method, *Group Locality*, is a source-to-executable, compiler oriented approach to execute tileable, partially memory centric code with simple or complex dependency pattern (T9.2). Both methodologies are discussed further below.

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[2]. It is used by NWChem to solve electrical Shrödinger equations.



✓ R0
✓ T1 Operator
R1
✓ T2 Operator
R2
T3 Operator
R3
T4 Operator
R4
✓ T Is Connected
✓ R Is

```
h1+ p2 } { g3+ g4 } |0>
g5 g6 ) * <0| { h1+ p2 } { g3
t ( p5 h6 ) * <0| { h1+ p2 } {
g3 g4 g5 g6 ) * t ( p7 h8 ) * <0
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
```

$$b^{h_7}_{h_1} = f^{h_7}_{h_1} + t^{p_8}_{h_1} a^{h_7}_{p_8} - t^{p_4}_{h_5} v^{h_5 h_7}_{h_1 p_4} - \frac{1}{2} t^{p_3 p_4}_{h_1 h_5} v^{h_5 h_7}_{p_3 p_4}$$

```
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;
            for(p4b = n
```

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 diagram gives a

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

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[3], 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.

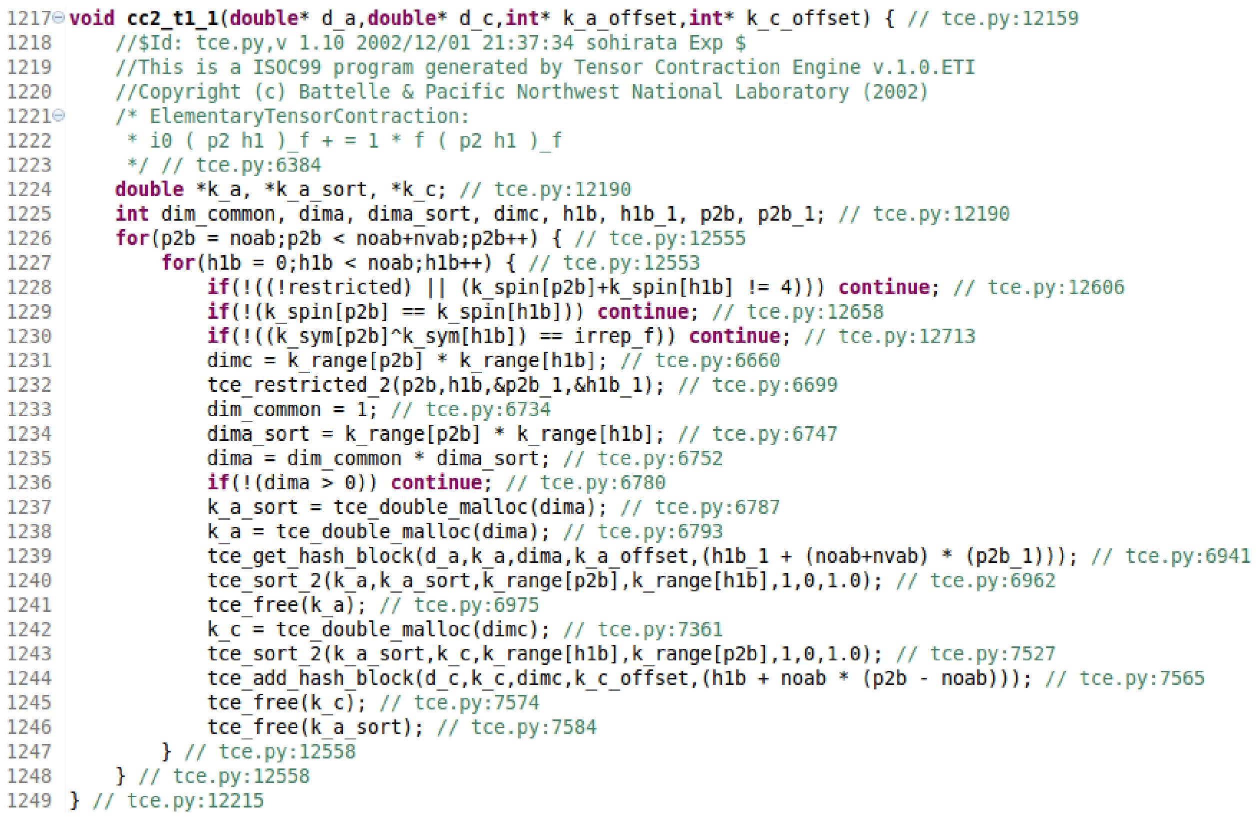
Proxy App

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:

[3] 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[4].

Availability

As mentioned above in the status section, ETI has begun to adapt the proxy app to run on task-based runtimes, and analyze its performance. Since the standalone serial version may be useful for other research teams, it is being made available for all to use. The latest version of the code, and some data sets to use with it, can be found on the DynAX website[5].

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

Reservoir: x86 virtual DMA experiments

This quarter, we modified the R-Stream SWARM backend to support calls to our x86 virtual DMA library. As detailed in the previous quarterly report, this library leverages SSE4 non-temporal loads and stores in order to avoid polluting the cache when loading and storing data to a virtual scratchpad.

A virtual scratchpad is a memory region that fits in the cache. R-Stream's virtual scratchpad optimization consists of pre-loading input data to a task into a virtual scratchpad, working on the copy of the data that sits in the virtual scratchpad during the task execution, and copying back out the task's output data. The main advantage of this technique is that virtual scratchpad data mostly stays cached during the execution of the task.

We compared the execution of programs parallelized to SWARM with and without virtual scratchpad implemented with x86 virtual DMAs. All codes were executed on a 12-core, 24-thread Intel Xeon CPU E5-2620 0 @ 2.00GHz. We implemented a machine model that defined the L2 cache as a scratchpad only accessible through explicit data transfers using our x86 virtual DMA library. All benchmarks were run 3 times with DMA and 3 times without DMA. The arithmetic mean of these results was used to calculate speedup.

A comparison between the average run times is reported in Figure 1. The same results are presented along with problem sizes in Table 1. The N parameter represents the basic size (matrix size, grid size, input signal size), while T represents the number of time steps used in the benchmark.

Benchmark	**Problem parameters**	**Percent Speedup**
Matrix-matrix multiply	N=1024	85.30%
Gauss-Seidel 2D 5 point	T=256, N=256	-3.95%
Gauss-Seidel 3D 7 point	T=256, N=256	7.03%
Jacobi 2D 9 points	T=1000, N=1000	97.91%
Reverse Time Migration 3D	T=2048, N = 256	426.30%
Filterbank FIR filter	N=8192, Filter=32, Banks=256	41.54%
FIR	N=8192, Filter=32	130.63%

Table 1. A description of the benchmarks, along with the percentage of speedup obtained w/ virtual DMA.

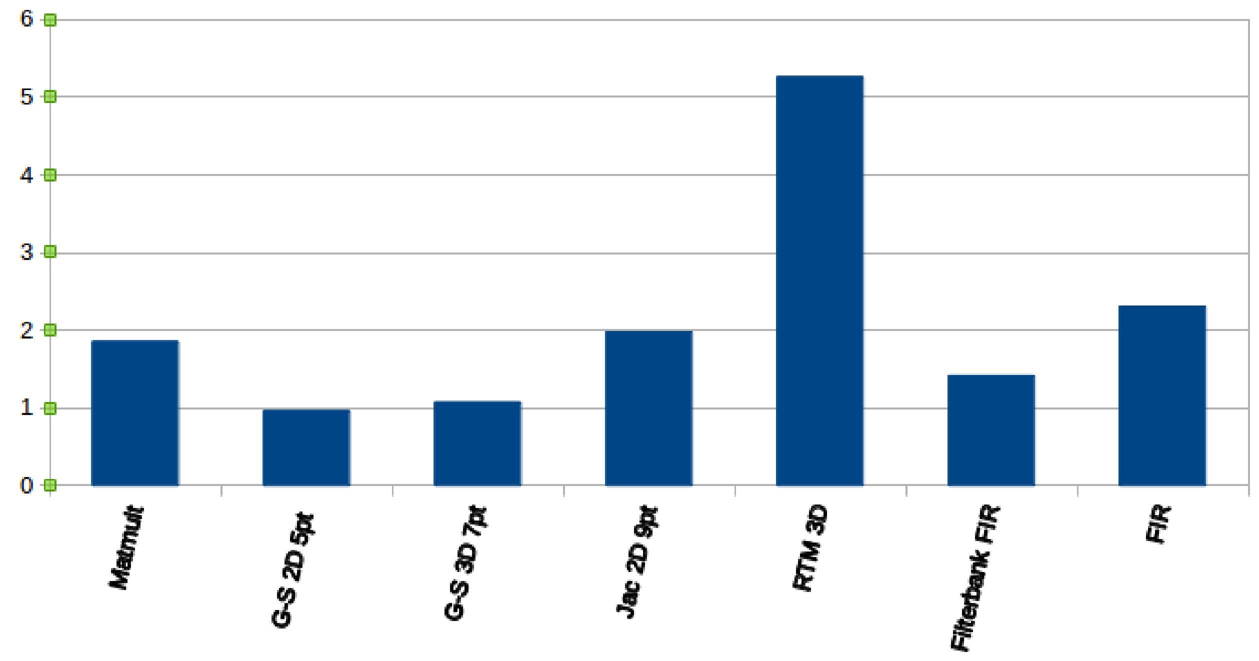


Figure 1. Speedups from virtual scratchpad w/ DMA

These preliminary results are extremely encouraging, since we haven't noticed a significant slowdown (yet) and speedups go beyond 5X for the Reverse-Time Migration benchmark. We believe higher performance could be achieved with further refinement and tuning of both the benchmarks (problem sizes) and DMA code generation.

The performance bottleneck in the Gauss-Seidel examples is not directly related to L2 cache usage. Otherwise, we do not see a straightforward interpretation of these results. There doesn't seem to be a direct relationship between the amount of reuse in the program and the obtained speedup. Some degree of correlation to the size of the benchmark is expected (and observed here, RTM 3D being the largest one in terms of problem size), but a larger data set is necessary before we can draw clear conclusions.

An interesting point about the non-filterbank FIR filter (whose code is reproduced below) is that one would expect the combination of hardware prefetchers and next-cache-line prefetching effects to cover most cache latencies. But in fact, since the loop becomes parallelized across the input data, it is likely that these hardware optimizations introduce undesirable cache coherence overhead by having one core fetch another core's data, as exposed by Zuckerman et al[6].

[6] S. Zuckerman, W. Jalby, "Tackling Cache-Line Stealing Effects Using Run-Time Adaptation."In proceeding of Languages and Compilers for Parallel Computing - 23rd International Workshop, LCPC 2010, Houston, TX, USA, October 7-9, 2010. Revised Selected Papers.

```
for (i=0; i<N; i++) {
   for (k=0; k<F; k++) {
      if (k<=i) {
         Y[i] = C[k] * X[i-k];
      }
   }
}
```

One consequence of the virtual scratchpad optimization is that the working data of each core becomes private to the core, preventing any form of cache-line stealing effect from happening.

In order to understand these speedups better, we would like to complete this preliminary study with the following experiments:

- Characterize the interference between DMAs and the processor’s hardware / next-line cache prefetchers by turning off these hardware optimizations.
- Directly compare virtual scratchpad without DMA against virtual scratchpad using DMA, in order to isolate the effect of the virtual DMA.
- Characterize the impact of using the virtual scratchpad for only the subset of data that has significant reuse within the task that works on it.
- Optimize away redundant transfers. The current communication generation scheme produces data transfers that could be eliminated.
- Leverage the explicit data movement as a way of performing sophisticated on-the-fly data transformations, for instance stencil-specific ones[7]. Some simple ones are already available in the current implementation.
- Look at speedups over a range of problem sizes.

The same optimization can be used to create virtual scratchpads associated with other memory entities, including NUMA banks associated with sockets. However, the different associated data movement costs may have different implications on the best choice of data sets to be copied. In the case of caches, loading data into virtual scratchpad seems to pose little risk, since data that is not reused is loaded into the cache anyways.

Finally, as in any new piece of software, we have observed a few bugs in the current implementation, which we will fix. The benchmarks presented here are not affected by these bugs.

[7] T. Henretty, K. Stock, L.N. Pouchet, F. Franchetti, J. Ramanujam and P. Sadayappan. "Data Layout Transformation for Stencil Computations on Short-Vector SIMD Architectures," International Conference on Compiler Construction (CC), March 2011.

UIUC: Design Changes Targeting the Distributed Memory Environment

High-Level Application Programming Interface

The programming interface is through the HTA library, which is similar to the HTA design proposed previously. The sequential part of the program is replicated to execute in parallel in different processes. Whenever HTA operations are encountered, processes acquire their own unique identifiers and execute different control flows to collaborate with each other in the computation. Most of the time, application programmers do not have to be aware of the SPMD model underneath. However, when they need to implement customized communication operations, they need to differentiate among parallel processes using process identifiers.

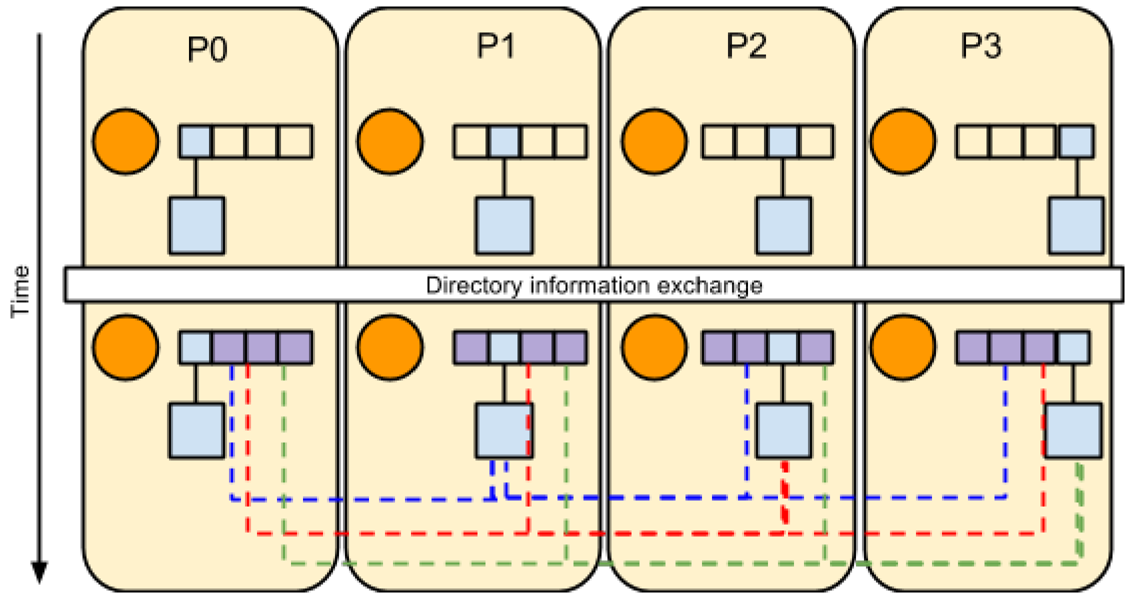
The communication and synchronization is a collaborative work among processes, which can only be done through specific HTA communication operations, which are the ones that cause data exchanges. HTA communication operations include shift, transposition, and some classes of assignment operations. For synchronization, we plan to provide explicit barrier operations and implicit barrier for some parallel operations.

HTA Design Changes for SPMD

Previously, the HTA library implementation was built under the assumption of running on shared memory SCALE. The design and implementation HTA need modifications in order to run on distributed memory SCALE. The most significant changes are the allocation of data and the communication operations.

In shared memory SCALE, data blocks are allocated once and subsequent accesses are done through pointers to the data blocks. But in the SPMD model, each process has its own address space, and thus it is not legal to access HTA tiles, or data blocks in general, owned by remote processes directly through pointers. Instead, HTA tiles are allocated in parallel when HTA_create() is called. HTA_create() computes the metadata part locally, and allocates the local portion of the data tiles. The tile reference information in each processor is communicated to all other processes through an allgather operation and stored in a local directory data structure everywhere.

The following figure demonstrates the allocation of an HTA. The orange circles are the tasks. When HTA_create() is called, each parallel allocation task allocates locally a directory (small squares) and a local data tile (big squares). Next, a directory information exchange by allgather is performed. After the exchange, each process now has remote tile reference information in its local directory (shown by purple squares and colored dotted lines).

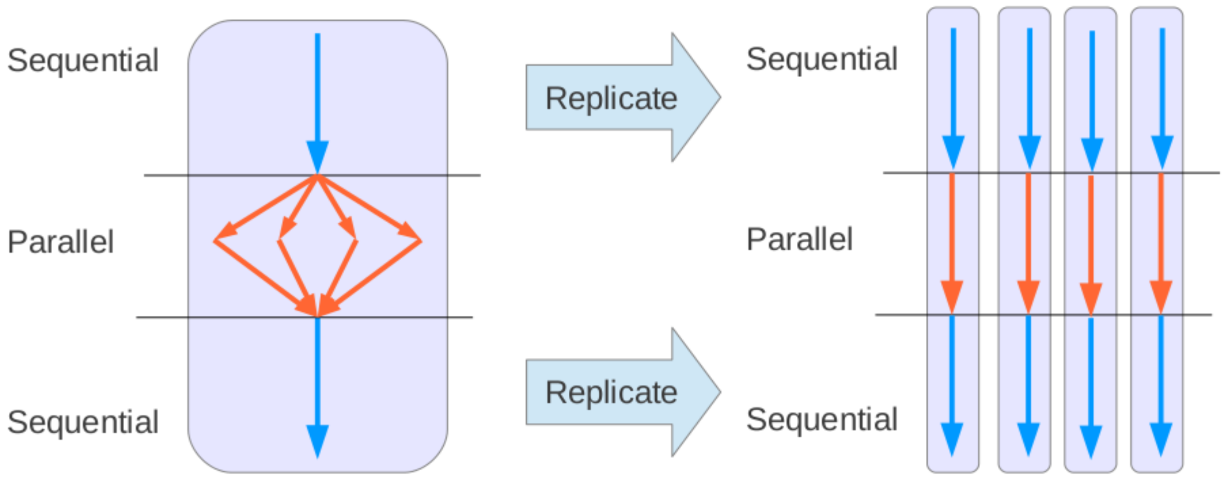


Parallel HTA Tile Allocation.

HTA communication operations are implemented using communication operations in the API provided by PIL. When a process needs to access a remote tile in an HTA operation, it can acquire the remote tile by first looking up in its local directory for the tile reference information and use this tile reference as function arguments for PIL communication operations. The underlying PIL implementation acquires the data tiles and ensures it is accessible locally before the subsequent operation starts.

SPMD Programming in PIL with Codelets

The PIL compiler generates SCALE code. Logically, PIL presents a different address space to each PIL node executing. Each PIL node is made up of a small collection of SCALE codelets. The codelets that are running can be in the same address space (shared memory) or a different address space (distributed memory). Each codelet is assigned an ID, similar to a rank in MPI. During execution, as the phase of the program switches between sequential execution and parallel execution, each PIL node for a single execution ID generates is successor node. During the sequential sections of the program, each node executes exactly the same instructions. Only during the parallel portions of the program can a node execute code and diverge depending on its ID. An example PIL program can be seen in the figure below. In the HTA programming interface, all HTA operations take place in the parallel sections of code, and all of the user code takes place in the replicated sequential sections of code.

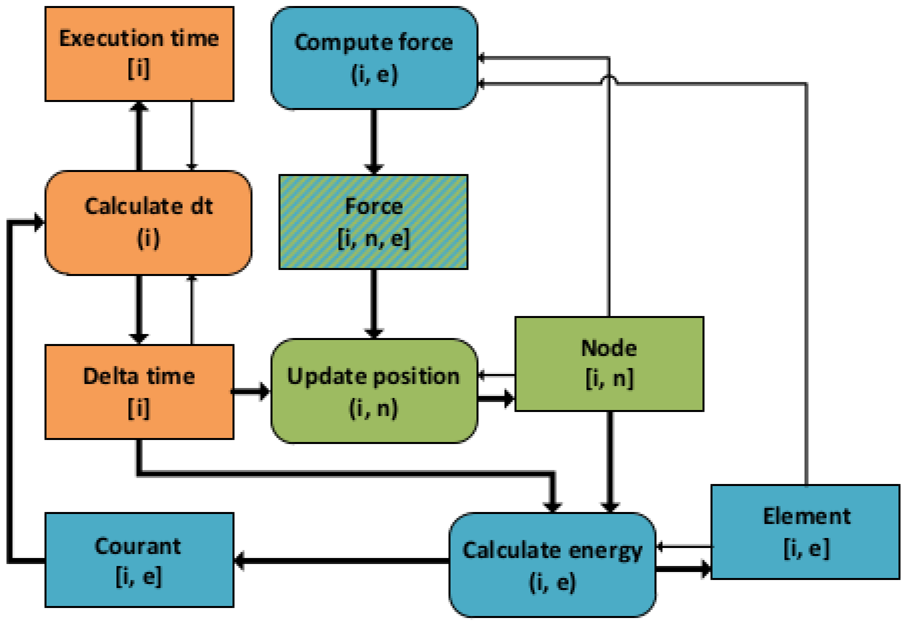


SPMD task address space representation for PIL nodes.

Currently we have implemented the generation and control of codelets in SCALE. We are still investigating collective operations, such as communication and synchronization. Since PIL can generate code for many different backends from the same source code, we must ensure that these collective operations in PIL are supported by all backends. We have various strategies for performing these operations in each backend, and are working on a unified theory of communication in PIL so that the programmer can have a single interface to all of the operations in all of the backends.

PNNL: Application and primitive data types

With respect to applications, we moved the code from C to C++, switched from a global data structure, to local data structures, and condensed all functions into 4 CnC steps. We then transformed the refactored code into CnC program using Intel's CnC framework and Rice's CnC-OCR framework. The Intel version is complete, tested, and running. The Rice version still has a few bugs. The final CnC diagram matching source code is below.



Architected Composite Data Types (ACDT) Framework

The *ACDT conceptual framework* was created to exploit opportunities that arise when the runtime is aware of a composite's properties (access patters, data composition, dynamic range, etc). This framework was designed with massive multithreaded environments in mind and uses the SWARM framework from ET International. We have conducted the following experiments (submitted for publication), adhering to the SOW's intent (T9.1):

1. We exercised the conceptual framework with two representative processing kernels - Matrix Vector Multiply and the Cholesky Decomposition applied to varying degrees of matrix sparsity. Both examples were coded using the SWARM programming / runtime model.
2. We opted for *compress/decompress engines* as transformations and as operators we chose *algebraic invariant operators*. We developed two different approaches based on transformation opaqueness in relation to the application.
3. We show that the two approaches have their strengths in HW or SW respectively, where the SW approach can yield performance and power improvements that are an order of magnitude better than implementations that are ACDT oblivious.

Group Locality (GL) Framework

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 operations. Similarly, using careful orchestration of memory accesses in form of data movement and data restructure management, interference in memory is reduced and accesses are made mostly contiguous. In order to achieve these goals we have developed the following methodologies (in process of publication submission) (T9.2):

1. Highly parallel tiling strategy with intra-tile parallelism. Such parallelism is designed to have parallel startup and execution for threads working together as a unit within a L2 (cache level) tile.

2. A Framework for fine grained parallelism where threads perform under a micro dataflow execution model. This framework currently uses PLUTO and CLOOG as a tool for tiled code generation.

3. Data movement and restructuring of data such that accesses with reuse are preserved in a state that matches the pattern of future access to minimize interference

Issues

ACDT Framework

A more in depth characterization of the ACDT type effects is needed to further describe the benefits and detriments of the compression engines and the framework in general. Moreover, the effects of different engines (and combinations of thereof) in a broader set of applications and workloads are needed.

GL Framework

Guarantying 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.

Goals for the Future

Applications

To date, we have considered tiling the Lulesh code, so each step is computed on one node or one element. It may make sense to combine neighbors to reduce CnC structure overhead. We plan to wrap-up the Lulesh work next quarter, writing a paper on our experience and describing the benefits of using CnC to design HPC parallel applications. Next quarter, we will begin investigating AMR.

ACDT Framework

Besides the proposed studies, reliability aspects of the application will play a larger part on the ACDT framework from hardware, software and application perspectives. Moreover, extra efforts on runtime time adaptation and different engines based on data patterns are planned for the next quarters.

GL Framework

The above-mentioned methodologies are an important element for achieving exascale performance aimed within the X-Stack project as they explore strong scaling aspects.
We continue working on improving our framework by developing experiments and techniques that will orchestrate data movement and avoid memory interference in memory banks and pages since such efforts can have direct and positive impact on both performance and power. Currently, we are enhancing the data re-structuring framework to be aware of the memory structure and we are going to test it in a many core platform.

Publications

Shrestha, S et.al, “Gregarious Tiling: A Novel Methodology for Collaborative Multithreaded Execution”, submitted to 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming in Florida, USA. (status: rejected; we are implementing peer-reviewer’s suggestions for resubmission)

Marquez, A. et.al, “ACDT: Architected Composite Data Types, Trading-in Unfettered Data Access for Improved Execution,” submitted to the 23rd International ACM symposium on High Performance Parallel and Distributed Computing 2014, Vancouver Canada.