High-level Status Summary

Technology	Description (Institution)	Status
Memory access semantics/Runtime	TCE mapping/porting to SWARM block level (ETI)	In Progress
Resilience	Containment Domains (ETI)	In Progress
Application migration	MPI Interoperability (ETI)	Evaluating
Parallelizing compiler	Support for distributed data and computation placement (Reservoir Labs)	Evaluating
Parallel Language	HTA library and PIL compiler design and implementation changes for SPMD execution (UIUC)	Done
Parallel Language	Evaluation of SPMD mode with NAS Parallel Benchmarks (UIUC)	In Progress
Applications	Lulesh refactoring (PNNL)	Done
Enhanced Data Types	Design of Composite Data Types (PNNL)	In Progress

Summaries of Quarterly Work (Q10)

A face to face meeting of Dynax project team has been conducted on Feb, 18th at ETI. During this meeting, we have reviewed the project progress overall as well as each individual team members. The progress report of Q10 is to follow.

ETI Work

During this reporting period (Q10: 12/01/2014-02/31/2014), ETI has been working on the following tasks, according to the SOW.

Task 8.1: Research integration of containment domain execution and recovery with codelet scheduling (in progress)

Task 8.2: Research integration of CD persistence infrastructure API with SWARM (in progress)

Task 10.1: Study MPI interoperability (in progress)

Resilience (containment domains)

This quarter, we made good progress in the improvements to our prototype implementation of containment domains in SWARM. Our initial version developed last quarter had many limitations, such as allowing only a single CD to be active at a time (across all threads), and allowing only one preservation per CD. We have enhanced our implementation so that these limitations no longer exist, allowing for greater flexibility on how CDs are placed in an application. We now also allow for nested CDs in a program.

In order to measure the efficiency of the containment domain approach, we have been adding containment domains to real applications written in SWARM. So far, we have done so with an implementation of Cholesky. The SWARM version of Cholesky exhibits very fine-grained parallelism, and uses no global barriers, so it is a good candidate to test the overhead of our CD implementation. Cholesky has three main tiled operations: POTRF, TRSM, and GEMM/SYRK (depending on tile location). We placed a CD around each of these, enveloping all of the actual calculation (in this case, calls to LAPACK). As a check function, rather than simulating hardware error reporting, we used a test generator (based on a random number generator) to check the success of a CD. Each of these CD types had an adjustable threshold, and if a random number was beyond this threshold, a fault was deemed to have occurred. The above has been completed, and we have started experimenting with tile sizes, matrix sizes, and failure rates to determine the overhead of our implementation. The results will be collected in the next quarter and analyzed.

We have kept in contact with leading containment domain researcher Mattan Erez from UT Austin, who has provided feedback on our ideas. During meetings with Prof. Gao, he has also suggested to us some directions for future research.

MPI Interoperability

One goal of the DynAX project is to facilitate interoperability with legacy MPI programs. We have considered multiple methods of accomplishing this goal, some of which are outlined below:

- MPI+SWARM: An MPI program with SWARM calls added
- SWARM+MPI: A SWARM program with MPI calls added
- Codelet MPI: Creating an MPI compatibility layer in SWARM
- Porting MPI to SWARM: Fully rewriting an MPI program in SWARM

Since the XPRESS team has similar goals, we have been in contact with them on this. We have reached out to Barbara Chapman at the University of Houston, who referred us to Alice Koniges at Lawrence Berkeley National Laboratory. We have discussed ways to work together towards this goal. Alice will supply ETI with a set of simple MPI+HPX benchmarks for us to understand how non-blocking MPI calls are used in this program. As described in the SOW, we rely on the XPRESS team to work with us to implement non-blocking MPI calls using continuations.

TCE

In previous quarters, we had completed a block-parallel implementation of TCE using OCR. We had also completed a task-parallel version using SWARM, with very coarse parallelism. This quarter, we started work on a block-parallel version using SWARM. This version is similar to the block-parallel OCR version, with a few optimizations.

SWARM

We created a new release of SWARM, version 0.17, on January 12. This version adds a workaround for an issue with idle threads, reported by UIUC. It is available for our partners to use in their respective work.

Reservoir Work

This quarter, Reservoir has contributed to the following SOW tasks:

Task 2.4: Research compiler code generation for data placement and movement

Task 3.4: Optimize unstructured computations

We furthered the development of an integrated compiler and runtime system that auto-parallelizes loop-nests to clusters. The auto-parallelization method separates the concern of finding parallelism in the computation to that of movement of data in the parallel computer: the compiler parallelizes code and inserts *logical* memory instructions that specify data that will be consumed and produced by the parallel task. The runtime is responsible for initiating and orchestrating communication by being cognizant of the underlying message passing mechanism. The logical memory instructions introduce a layer of abstraction from the distributed data, and tasks follow a get-compute-put pattern, which makes our code generation compatible with PNNL's PEDAL scheme, in which the communication layer is responsible for runtime optimizations. As explained before, this communication layer is a necessary step towards Task 3.4, as it will have an important role in the automatic parallelization of block sparse codes.

Automatic Cluster Parallelization

The generated cluster-parallel programs have the SPMD (Single Program Multiple Data) form. Data communication between processors is performed at the granularity of tiles: tiling program transformation aggregates loop iterations in such a way that each tile can execute atomically and communication between processors occurs at tile boundaries, if necessary.

The communication operations are abstracted as logical DMA (Direct Memory Access) primitives --- each tile issues logical DMA GETs to fetch data needed for computation and

PUTs to store data produced by the tile. The logical DMAs are unaware of how the data is distributed on the distributed memory of the cluster and where the requested data is located.

The logical DMA operations are in turn implemented as an R-Stream runtime layer functionality using the Global Arrays toolkit. Global Arrays (GAs) provide a global address space for creating and accessing arrays. We illustrate the cluster-parallelization in R-Stream using an example. Consider the loop shown below:

```
int i;
for (i=0; i< N; i++) {
  A[i] = B[i] + 1;
}
```

It adds a constant --- 1 to N elements of array B and stores the result in array A. The R-Stream compiler for this input and a 4-node cluster, produces the following parallelized code:

```
int PROC = r_nodeid();
float A_l[N/4];
float B_l[N/4];
r_dma_get(B_id,(N/4)*PROC,B_l,1,1,N/4);
int i;
for (i = 0; i < N/4; i++) {
    A_l[i] = B_l[i] + 1;
}

r_dma_put(A_l,A_id,(N/4)*PROC,1,1,N/4);
```

The computation is tiled such that each processor increments N/4 elements of array B. A DMA GET instruction is issued for the data required by the tile and the data written in the loop are stored using a DMA PUT operation at the end.

The DMA engine in R-Stream emits efficient logical DMA operations: whenever the data to be read and written have contiguous portions in them, data movement will be orchestrated in such a way that there will be a single DMA call for a contiguous segment of data.

R-Stream Runtime Layer

The creation of data structures in the global address space and implementation of logical DMA calls generated by the compiler are handled by the runtime system. We now present the runtime API.

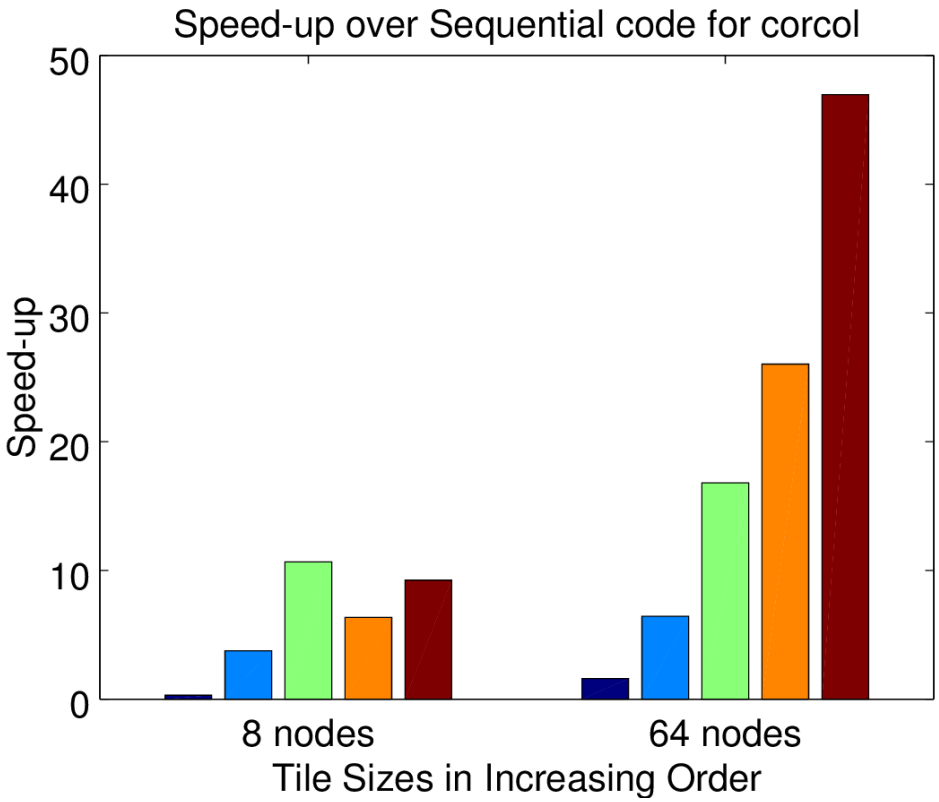
```
void r_create_array(int data_type, void *src, int size, int ga_id);
void r_copy_array(int ga_id, void *dst);
void r_dma_get(int ga_id, int index, void *dst, int src_stride, int dst_stride, int count);
void r_dma_put(void *src, int ga_id, int index, int src_stride, int dst_stride, int count);
int r_nodeid();
int r_nnodes();
void rga_sync();
```

An invocation of *rga_create_array* function creates a global array --- array in the global address space. The datatype of the array elements and array size have to be specified. Additionally, the compiler associates an identifier --- ga_id for the array. Whenever a DMA operation has to be performed on the array, it refers to the array using this identifier. The global array thus created is populated with data from the buffer pointed to by the *src* pointer. The *rga_dma_get* and *rga_dma_put* functions implement logical DMA GET and PUT functions respectively. They require as input: the global array identifier, source and destination buffer pointers, source and destination strides, and finally, the number of array elements to be accessed. Function *rga_nodeid* returns the rank of the process; *rga_nnodes* can be used to get the number of processes; barrier functionality is realized through *rga_sync* function.

Experiments

We performed preliminary experiments using the *corcol* benchmark which computes a correlation matrix. The problem size was set to 3000 X 3000. The codes were cluster-parallelized using R-Stream and were compiled with PathScale (tm) Compiler Suite: Version 4.0.10 and -O3 flag. The resulting binaries were run on 8 nodes and 64 nodes of a cluster. Each node is made up of AMD Opteron(TM) Processor model 6272 processors. One process was launched per node as inter-node parallelization is the focus of this experiment.

The following figure shows the speedup achieved by the auto-parallelized *corcol* over sequential *corcol*.



We explored five different tile sizes. The graph shows speedups of those five tile sizes. The tile sizes are in increasing order. When the parallel program is run on 8 nodes, the third largest tile sizes achieves the highest speedup, while the largest tile size performs best when the code is run on 64 nodes.

Publications

We are working on a publication that presents the cluster backend developed within DynAX, as well as a data redundancy technique to reduce inter-node communications (developed within a separate context but available in R-Stream). We plan to submit this work to ROSS 2015, the International Workshop on Runtime and Operating Systems for Supercomputers, mid- March. This paper will show how our logical communication layer enables the study of data placement and movement (Task 2.4) at the cluster level.

Future work

For the next two quarters, we will continue our effort to support the automatic parallelization of block sparse codes through our cluster backend and runtime layer. We will also continue to improve other aspects of the SOW as necessary.

UIUC Work

In this quarter, the UIUC team focused on:

Task 5.3': Evaluation of the PIL implementation and API (in progress)

SPMD Execution Mode

In the last quarter, we completed the design of SPMD PIL programming interface and the compiler backend for OpenMP. This quarter, we implemented the compiler backend for shared-memory SCALE and the components of the HTA library needed to make use of the SPMD mode of PIL.

Since, in SPMD mode, it is important to ensure different "processes" run asynchronously as much as possible to reduce unnecessary waits, we rewrote part of the HTA library to remove the barrier that the prior version of HTA executed after each array operation. We are planning to further improve our shared memory implementation of HTA to take advantage of fast synchronization primitives and also avoid unnecessary data-copying. We intend to explore these shared-memory optimizations and observe the performance impacts.

We have finished the implementation of shared-memory SCALE backend for SPMD PIL and the HTA library for SPMD PIL. Necessary changes in NAS Parallel Benchmarks (EP, IS, FT, MG, CG, LU) were also made to execute in SPMD mode. We expect to deliver a complete performance evaluation comparing fork-join PIL, SPMD PIL with different backend runtime systems (OpenMP, SCALE, OCR) using NAS Parallel Benchmarks in the next quarter. We also plan to implement algorithms (such as tiled LU factorization and Cholesky factorization) that benefits from asynchronous parallel execution.

Future Work

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

PNNL Work

Task 9.2: Investigate initial and static data placement at each memory models

Task 9.1: Research multi layered power optimized data representation

During this quarter, PNNL concentrated on the compiler and runtime aspects of our research. This was exemplified by porting the Group Locality framework to a new many core platform to conduct memory experiments. Here we focus on the extension of Group Locality concepts to a many core architecture --Tilera's Tile GX -- (Task 9.2).
Furthermore, we emphasized the development and analysis of the distributed ACDT framework on an asynchronous runtime at scale. (Task 9.1).

Group Locality(GL) (Task 9.2)

The porting of the initial jagged framework has been completed and it is being enhanced with memory restructuring to extract more parallelism and locality from the machine. The new experimental testbed is the Tilera's TileGX platform (see Figure 1) which salient features that include 36 VLIW cores, 8 GiB of DRAM, distributed L3 cache across the cores, fine grain data placement and control and 5 distinct physical mesh networks connecting all the cores. This platform provides a stable sandbox to perform memory analysis and memory restructuring studies for our Group Locality (GL) framework.

Figure 1: TileGX Many Core Architecture

The Group Locality framework has the concept of the re-structuring buffer in which data structures will be modified into more suitable access patterns for the group of threads working

on it. The idea is to use these spatial and temporal schedules for the fine grained GL runtime system to further extract performance out of the applications. The restructuring can be influenced from a thread perspective (e.g. access patterns) or from a memory hierarchy perspective (e.g. cache / memory configuration).

Architected Composite Data Types (ACDT) (Task 9.1)

In the effort to port the ACDT framework to exascale targets, we are analyzing and developing an ACDT version that will run on a distributed system at scale. We are using SWARM as our base asynchronous runtime system and Cholesky as our leading example. We are conducting a scalability study on the current SWARM code using ranges up to 2048 cores (128 nodes with 16 physical cores). Moreover, we are analyzing the current distributed Cholesky code to find places in which ACDT can show an important advantage.

The scalability study and the code analysis is a work in progress and it will require a very close interaction between PNNL and ETI.

Publications

Resource Aware Concurrent Start for Stencil Applications
Sunil Shrestha, Joseph Manzano, Andres Marquez, and John Feo, and Guang R. Gao at 2015 International Symposium on Code Generation and Optimization, San Francisco, February 7-11

Future Work

We will continue our research on Tasks 9.1 and 9.2: In particular, we will assess, similar to the work that we did on the Intel Phi, if specific Tilera architectural artifacts hamper our GL framework and how to remedy this artifacts.
Our ACDT work on SWARM will go through a similar process. We will need to first establish a clean, distributed SWARM non-ACDT baseline to perform valid comparisons with the ACDT version to exclude any artifacts deriving from the machine or the runtime system itself. Eventually, we will also need to evaluate which ACDT enhancements recently made on OCR can be transliterated to SWARM.