Critical Technology Evaluations

Technology	Description	Status
CoDesign App	Converting NWChem Tensor Contraction Engine to C for use in XStack projects	Evaluating
CoDesign App	Converting LULESH to codelet execution model	Evaluating
Parallel Language	Sparse data representation in PIL	Completed
Parallel Language	Evaluation of PIL targeting Distributed SCALE using NAS Parallel Benchmarks	Identified
Compiler	Optimization of automatic single-node parallelization to SWARM	Evaluating
Compiler	Polyhedral mapping scalability improvement	Evaluating
Compiler	Multi-node data placement and movement	Evaluating
Power Efficient Data Abstraction Layer	Started a theoretical framework called "Group Locality" that will be used to identify opportunities for data manipulation such as compression	Evaluating
Power Efficient Data Abstraction Layer	Survey of state-of-the-art compression algorithms	Evaluating
Power Efficient Data Abstraction Layer	Working on 2 measurement frameworks	Evaluating

Summaries of Quarterly Work (Q4)

ETI Work

TCE

ETI has begun to look at the Tensor Contraction Engine code generator, provided by PNNL. This takes user-defined tensor equations, and generates Fortran code to evaluate the equations. We have studied the generator, the Fortran code, and the environment that code executes in.

We have begun working on a method to generate equivalent C code, and are currently working to verify the correctness of this generated code. Once we have verified the C code is being generated correctly, we can begin to apply optimizations and scalability improvements to it.

LULESH

We spent some time studying the LULESH code. At the top level, the code iterates until the iteration limit is reached. If no iteration limit is specified, it runs until the shock wave reaches a radius of 1.0 distance units, which happens at the 932nd iteration. In our testing, an OpenMP run with no iteration limit takes about 9 seconds on a single 12-core compute node. There are command line parameters to artificially increase the amount of work per iteration, by duplicating some of the calculation.

We built a serial version of the application and profiled it using kcachegrind. The profile indicates that computation is split across 6 major areas, the largest of which occupies 23% of the overall time. We focused on the largest function (EvalEOSForElems) first. By rearranging loops and merging them together, we were able to show that many of the intermediate values could be simple scalars that only exist within a single iteration. This allowed us to decrease the memory footprint, and allowed the computation to fit into cache better. This initial set of optimizations reduced the overhead of that function to 68% of its original overhead, and reduced the overall program execution time by about 7%.

Our general plan of attack is to make the serial code as fast as we can make it, and then find the best way of parallelizing and distributing it using SWARM.

Reservoir work

Regarding the development of the compiler, our efforts were focused on three main aspects: producing SWARM code that has better scalability properties, increasing the tractability of R-Stream's mapping process, and introducing distributed data placement.

We were able to test the limits of our initial implementation of a mapper from sequential C to parallel SWARM in terms of its scalability on a single node, and refined our execution model into a more scalable one, as presented in section "Optimization of automatic single-node parallelization from mappable C to SWARM."

We also worked on improving the tractability of the mapping process itself, i.e, improving compilation time. There are two ways in which we plan to tackle this issue. The first effort, started in this quarter and presented in section “Improving scalability of polyhedral core engine,” is to replace expensive polyhedral computations made in Reservoir’s core polyhedral engine, “Jolylib,” with computations that are less expensive within the set of polyhedrons whose number of facets are small relative to their number of vertices. The second method we will use to address this issue is to enable scalable hierarchical mapping by containing the dimensionality of the problem across levels of the hierarchy.

A key component of our ability to parallelize codes to distributed systems is the management of distributed data. We are developing support for creation of bulk data movement through arbitrarily distributed and laid-out data. The description of this component is proprietary and hence will be disclosed outside of this report.

UIUC Work

In many parallel computing applications, there exists the need to store and compute with large sparse data. In many cases, the non-zeroes in a sparse matrix are only a small percentage of the total number of elements. When the data is sparse, it is often inefficient to store and compute using regular dense format. For example, concerning a matrix multiplication using a sparse matrix as an operand, many scalar multiplications do not have to be performed if one of the operand is a zero.

The computation involving zero-value elements can be written in a more efficient way without having to actually read them from memory. Since it is not necessary to read them, they do not need to be stored in memory either. It is desirable to provide specialized operations that exploit the sparsity and omit unnecessary computations, and it is also beneficial to express sparse data in a way that zero-value data elements do not occupy memory space. However, it is more complex if programmers have to define their own representation and tailor specific operations for the representation while dealing with sparse data. A better approach is to provide a library to facilitate sparse data manipulation.

During the last quarter we added to PIL the support for sparse data representation by extending the PIL HTA library. Programmers can now create sparse arrays in PIL code and use common operations provided as library routines. The sparse array implementation was used to implement the CG benchmark, a conjugate gradient method application from NAS Parallel Benchmarks (NPB), for execution on SCALE.

PNNL Work

We provided ETI with Tensor Contraction Engine code generator from NWChem. The generator translates tensor equations written by scientists to Fortran code. ETI is developing a generator for C code.

This quarter we tested several different approaches for fine grain execution as a target for high level compiler analysis. We have developed a couple of approaches for tiling fine grain execution. In the first approach, a permutable band (i.e. a collection of tiles that can be run concurrently as defined in the literature) is created. This approach allows us to better utilize hardware resources in manycore systems that make many hardware threads are available. The second approach, which we refer to as “execution band,” highlights inter- and intra- tile

dependency allowing dependency percolation with minimal overhead. It is similar to a dataflow computation. **Very preliminary** experiments confirm the promise of both approaches. The figures below display preliminary numbers and a comparison with OpenMP allowing us to identify important opportunities for PEDAL. Figure 1 compares execution for different numbers of thread teams (X axis) versus time (Y axis) for input sizes 2K through 32K. The figure shows the decrease in execution time as more thread groups work on larger arrays. Smaller arrays also benefit from the extra threads, but the benefit is offset by data starvation.

Figures 2 and 3 show that the fine-grain threading model is at least as good as OpenMP for 8K and 16K square arrays. Currently, we are characterizing the application and architecture to identify further optimization opportunities. We are working on a publication on the characterization of fine grained execution model on current designs.

Our Valgrind based tool has advanced toward the visualization and analysis of data. It can now express Codelets' coarse characteristics (communication, maximum concurrency expressed, access pattern) and memory properties. However, more analysis and work is needed to support more levels of details, such as accurate timing instead of partial ordering.

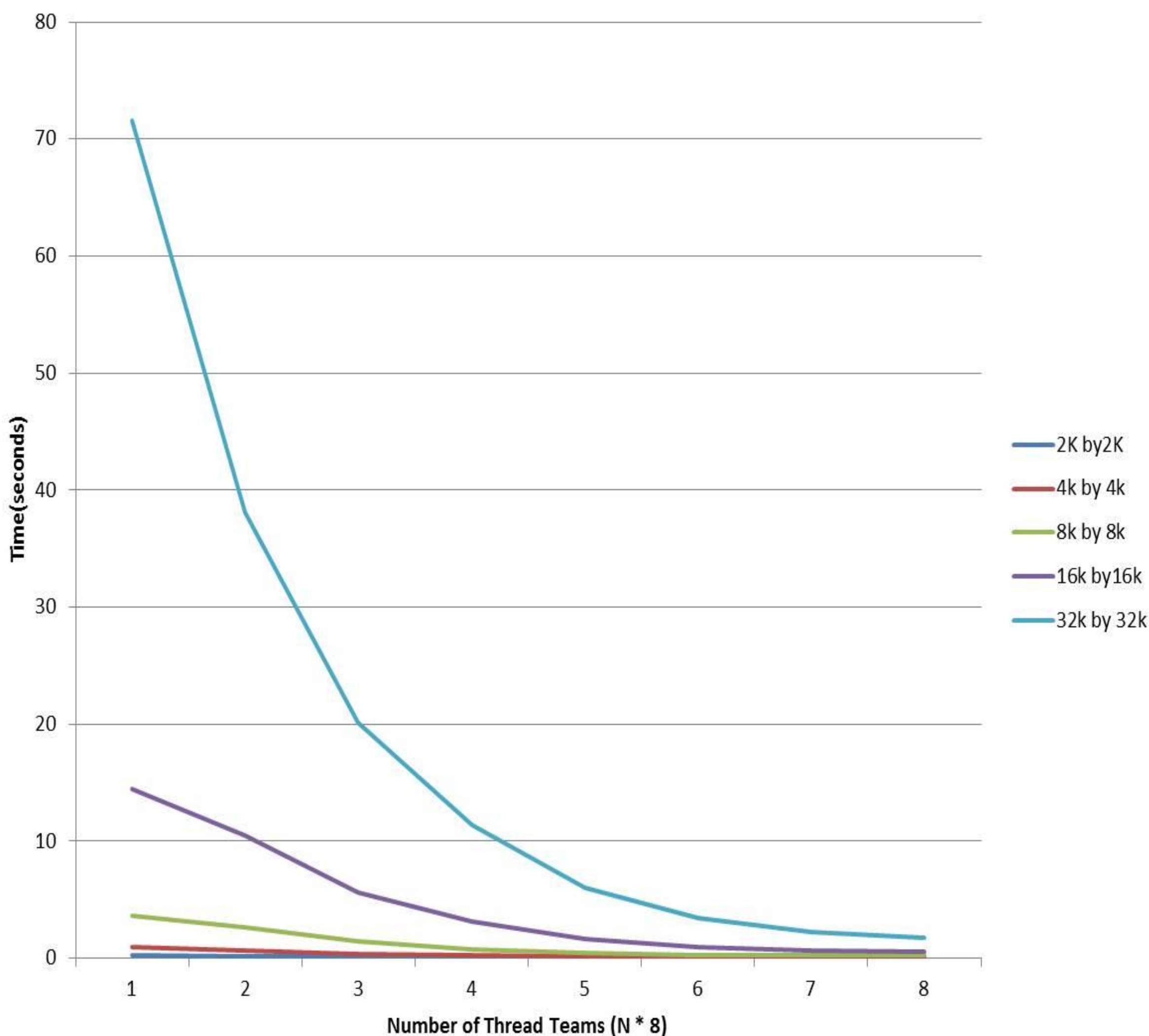


Figure 1: Comparison of number of thread teams for array sizes.

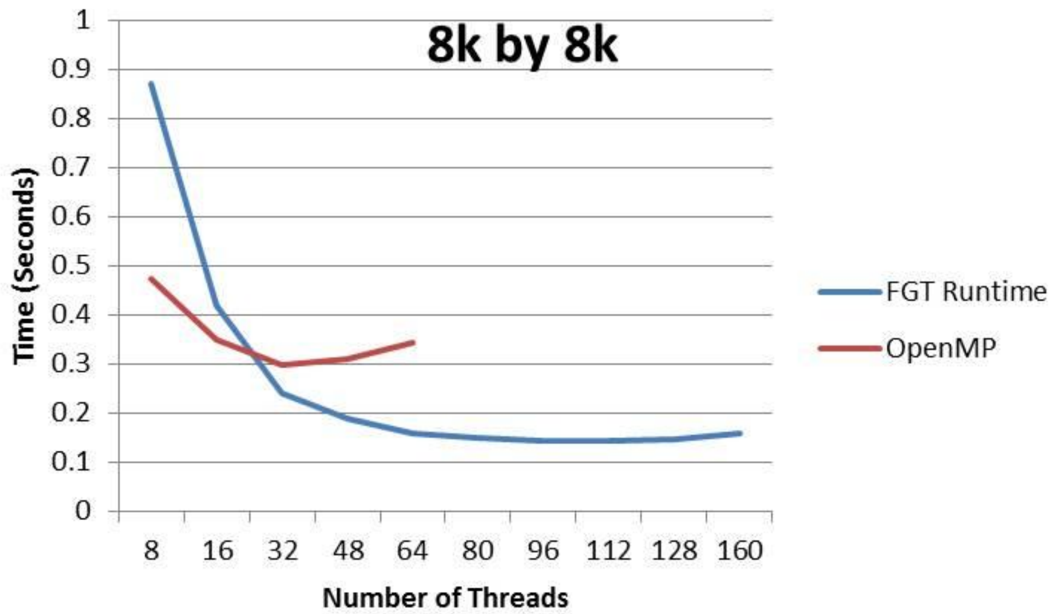


Figure 2: Runtime of OpenMP versus Fine Grained Tiling Framework with 8K by 8K Matrices

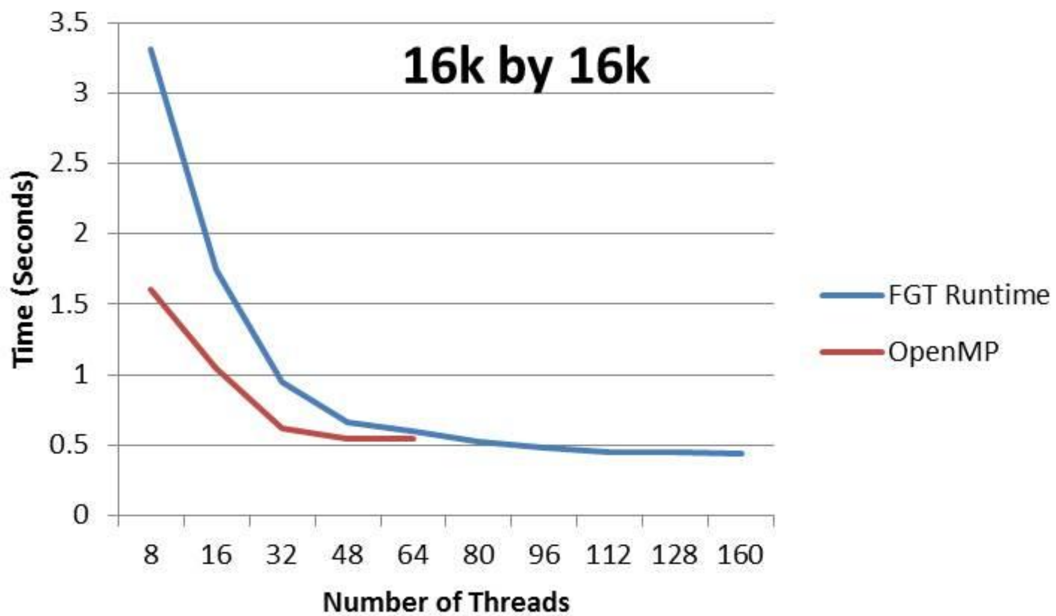


Figure 3: Runtime of OpenMP versus Fine Grained Tiling Framework with 16K by 16K Matrices

Topic Detail:

ETI: Tensor Contraction Engine C Generator

The application

The Coupled Cluster system provided by PNNL generates code from tensor equations supplied by the user. We were given the code generator (implemented as a Python library), and some sample input tensor equations and output Fortran code. The Python library performs optimizations on the tensor equations, reducing duplicate computation and selecting an optimal order of operations. It then generates scalable code to implement the resulting equations. The generated Fortran code uses Global Arrays to manage execution across multiple nodes, and the input and output matrices are partitioned into blocks of file data on disk.

The code generator library can optimize the tensor equations in the following ways:

- Contraction of a creation and an annihilation operator
- Identify equivalent terms and merge them together
- Erase terms that will lead to vanishing contributions

Also, the generated Fortran code emits calls to BLAS functions in some cases, and allows the computation to be striped across multiple processes and/or compute nodes.

In pseudo-code, a set of input tensor equations may look like this:

```
For all p2:  For all h1:
  i0(p2,h1)  =  1 * f(p2,h1)
  For all h3:
    i0(p2,h1) += -1 * f(h3,h1) * t(p2,h3)
  For all p3:  For all h4:
    i0(p2,h1) += -1 * t(p3,h4) * v(h4,p2,h1,p3)
    i0(p2,h1) += -1 * t(p3,h1) * t(p2,h4) * f(h4,p3)
  For all h3:  For all p4:  For all p5:  For all h6:
    i0(p2,h1) += -1 * t(p2,h3) * t(p4,h1) * t(p5,h6) * v(h6,h3,p5,p4)
```

After optimization, the same operation looks like this:

```
For all p2:  For all h1:
  i0(p2,h1)  = 1 * f(p2,h1)
  For all h7:
    For all p8:
      i2(h7,p8)  = 1 * f(h7,p8)
      For all p5:  For all h6:
        i2(h7,p8) += 1 * t(p5,h6) * v(h6,h7,p5,p8)
      i1(h7,h1)  = 1 * f(h7,h1)
      i1(h7,h1) += 1 * t(p8,h1) * i2(h7,p8)
    i0(p2,h1) -= 1 * t(p2,h7) * i1(h1,h7)
  For all p3:  For all h4:
    i0(p2,h1) -= 1 * t(p3,h4) * v(h4,p2,h1,p3)
```

We have added a C generator code to the TCE Python library, and are currently verifying the correctness of this output. The generated code has this C prototype:

```
void cc2_t0(int d_f1, int d_i0, int d_t1, int d_v2,
            int *f1_offset, int *i0_offset, int *t1_offset, int *v2_offset)
```

The inputs and outputs of the generated code vary depending on user input. The code generated for the above operation takes 3 arrays as input, and produces 1 array of output. Each array is represented as a file handle (e.g. d_f1) denoting which file the data lives in, and an offset table (e.g. k_f1_offset) denoting where each block of partitioned data sits within the file. These values are passed to I/O functions, to access the data when needed.

For the above operation, the input arrays are:

- f1 (2-dimensional fock matrix)
- t1 (2-dimensional cc-amplitude matrix)
- t2 (4-dimensional cc-amplitude array)

The output array is:

- i0 (2-dimensional)

The optimizer allocated two temporary arrays for intermediate values:

- i1 (2-dimensional)
- i2 (2-dimensional)

Future work

We are currently working to ensure the correctness of the generated C code, and have not yet started our analysis of that code's performance. However, some things are already apparent to us from having analyzed the Fortran code:

- The code, as generated, is performing all of the tensor equations in sequence. One equation (which generates one fortran subroutine) is completed before processing begins on the next. It is possible, and probably advantageous, to assign tiles of output to compute nodes and keep the partial output matrix in memory, reducing the amount of intermediate data written to disk.
- The workload balance (or imbalance) is completely dependent on the partitioning of the input file, and how well that applies to the node count. Each generated function has a barrier at the end, preventing threads/nodes from working on the next thing until the previous thing is complete. The functions are almost completely independent of each other, and as long as the output is summed together correctly, and the intermediate data is generated before it is used, the barriers are largely unnecessary. If they produce significant workload imbalance, we will use dynamic scheduling to reduce dead time between executions.

This list is not comprehensive. There are also areas where copying can be reduced, or done more efficiently, and other micro-optimizations along those lines. Generally, we expect that runtime analysis of the code's performance should indicate the areas where optimization work will produce the most benefit.

Reservoir: performance numbers for single-node parallelization

We ran scaling performance tests on a 16-core, 2 socket 2x hyperthreaded Xeon CPU E5-2690 0 @ 2.90GHz server on a selection of benchmarks.
The goal of this experiment is to estimate success of the current mapping strategy in codes that are small enough for a multicore x86 machine. Preliminary results are plotted on Figure 4.

The stencil codes based on jacobi and gauss-seidel were given tile sizes of 16 except for their innermost dimension, which was set to 64. The matmult benchmark was given a tile size of 64x32x32. Beyond this rough tile size selection, no particular optimization was performed to the code. The set of cores used for the runs were controlled using *numactl*, with an interleaved memory allocation strategy and usage of cores 0 to (n-1) where n is the number of SWARM threads, defined using SWARM_NR_THREADS=n.

The size of the codes were not tuned to the machine however, which explains that a few codes (e.g., sor, jac2d, trisolv, adi, fsymm) don't expose much scaling because they are too small. Also, these results are not smoothed in that they represent only one run of each code.

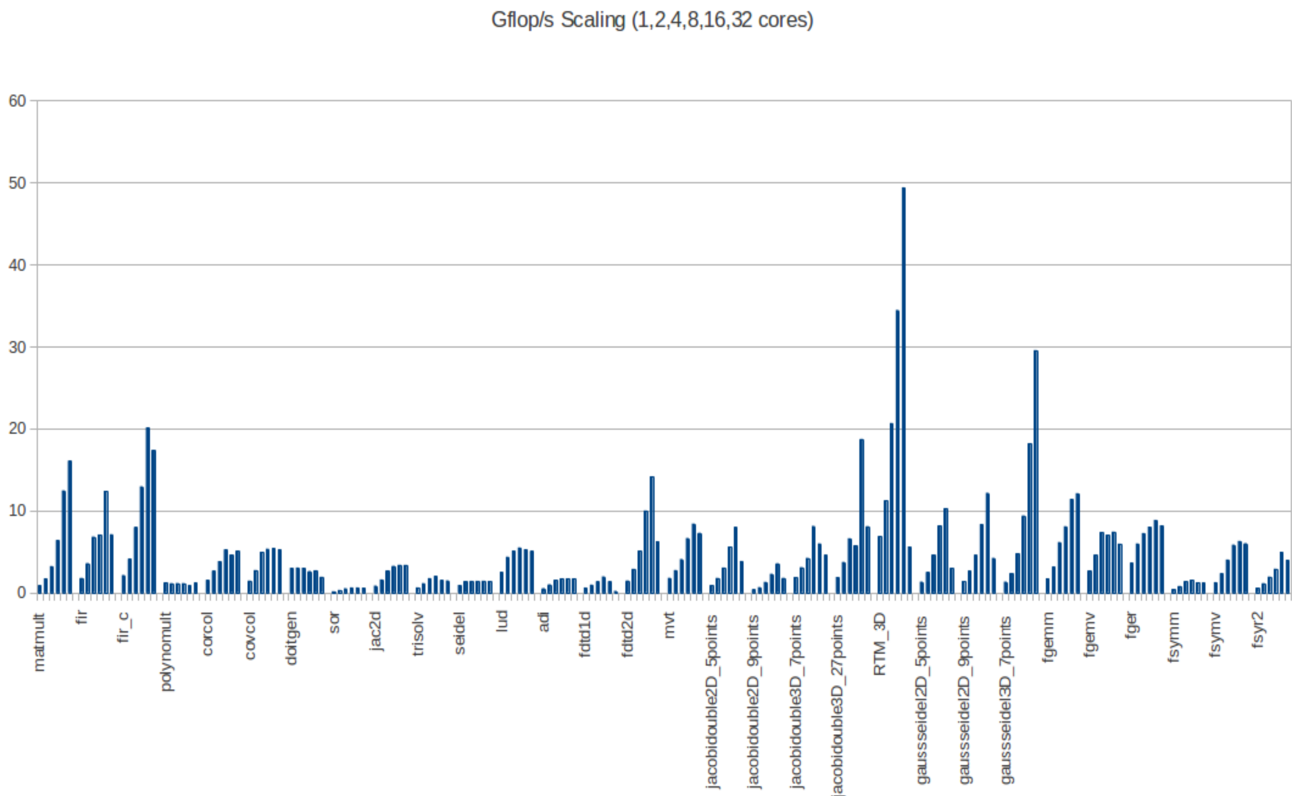


Figure 4. Preliminary scaling tests.

Globally, the execution seems to scale very well up to 16 threads. For several benchmarks, however - although it is not general - the use of hyperthreading (to get to 32 threads) translates into a performance dip. This could be happening for several reasons.

One reason is that competition for resources with the operating system is largely increased when all the physical threads are solicited by one process (our benchmark). Several preemptions from the operating systems can delay the execution of codelets. This delays

other codelets, but it also modifies the way the remaining codelets are scheduled, which impacts the locality of data accesses. This effect is aggravated in our measurement by the fact that we only sampled each run once (we will present smoothed data in the next report).

Another reason could come from the codelet's memory footprint being appropriate for L1. With benchmarks in which adjacent codelets don't share a lot of data (such as all the stencil codes or the parallel FIR filters), using two threads concurrently on the same L1 cache would then largely increase the L1 cache miss rate. A solution to this would be to modify the codelet memory footprint (by making R-Stream reduce its tile sizes during the parallelization process) to reduce cache interferences between hyperthreads.

Part of this effect could also be due to increased contention on the tag table. This sounds unlikely however in the stencil cases, since the number of bins used for synchronization keeps the number of tags using the same bin to a small constant (the number of loop dimensions in the case of the stencil). In the next section, we discuss other effects of using tag tables on scalability to very large number of codelets

Reservoir: Optimization of automatic single-node parallelization from mappable C to SWARM

Current Implementation

In order to make use of SWARM as a parallelization target, we defined a canonical way of producing parallel programs and chose a subset of the features of SWARM to implement it.

R-Stream currently forms codelets which first issue asynchronous "gets" that correspond to their input dependences. Each of these "gets" checks for the presence of a specific tag in a SWARM tag table, and re-schedules the codelet upon availability of the tag. When all of the input tags are present, the codelets perform their work (for instance solving iterations a PDE on a portion of the data). When done working, they put tags that correspond to their output dependences in the SWARM tag table. R-Stream uses ranking functions to define a unique tag per pair of codelets related by a dependence relationship. All codelets that will run in a mapped function are scheduled at the beginning of the function.

This system ensures that tasks can run as soon as their input dependences are satisfied. SWARM's asynchronous "gets" also ensure that no resource is used to wait (for a tag) while the resource could be used for computing.

In order to test the scalability of this existing way of mapping sequential C to SWARM, we had R-Stream decompose sample programs into high numbers (millions) of very small codelets. We did not use any method for removing transitive dependences. In one example, 35 million codelets were created. Memory occupancy during the execution of this example program rapidly went to 100% and the program was eventually killed by the operating system.

This behavior is understandable. Given a large number of transitive dependences where one tag is defined for each dependence, almost a quadrillion tags could theoretically be stored or in waiting in the tag table. While the number of tags was likely not this high for this example, even the few billion which were reached in this case are enough to saturate the memory of a current computer.

The main reason for this memory blowup is the high potential number of tags that are concurrently active (in $O(n^2)$where nis the number of codelets), which is then aggravated by the fact that all codelets are scheduled (and hence may issue their asynchronous “gets”) at once.

More scalable implementation

We can both reduce the number of active synchronization objects and the number of simultaneously scheduled codelets by replacing tag tables with SWARM’s *counted dependences.* A SWARM counted dependence associates a counter with a codelet. Counted dependences can be incremented or decremented atomically by any amount. When the counter reaches zero, the codelet is scheduled.

For each codelet, the new system - currently under implementation - will count the number of input dependences, initialize a counted dependence with this number and associate it with the codelet. These initializations will be performed in parallel for all the codelets, by codelets called “prescriber codelets”.

When a codelet is done running, instead of putting tags, it will decrement each of its successor codelets’ counted dependences by one.

As a result:

- onlyO(n) synchronization structures need to be created, and
- codelets are scheduled only when their input dependences are satisfied, which reduces scheduler overhead.

The number of synchronization structures active at the same time can be further reduced in several ways. Let us show how by first exposing a problem in the system we just defined, and present a solution for it.

The problem is simply that with the system we just presented, a codelet that is done working may want to decrement the counted dependence associated with a codelet whose counted dependence wasn’t initialized yet (because its prescriber codelet hasn’t run yet).

The fix requires a modification of the counted dependence decrement procedure. We replace the simple decrement of the counted dependence D associated with a successor codelet C with the following procedure:

```
1: if D is uninitialized (atomic test)
2:    count M = number of input deps for C
3:    initialize D’(M, C).
4:    atomically: if D is (still) uninitialized, set it to D = D’.
5: atomically decrement D.
```

Similarly, the prescriber codelet that was originally initializing the counted dependence must be aware of this. Instead of doing:

```
1: M = number of input dependences for C
2: initialize D(M, C)
```

It will do:
1: if D is uninitialized (atomic test)
2:    M = number of input dependences for C
3:    initialize D'(M,C)
4:    atomically: if D is (still) uninitialized, set it to D'=D.

While the potential number of temporarily initialized counted dependences is larger with this method, it also has one additional advantage: most of the prescriber tasks become unnecessary. In fact, if S is a dominant set of codelets, i.e., all codelets in the program have an ancestor in S, then it is only necessary to schedule prescriber codelets for S. This brings down the sequential overhead of starting the codelets from n to a fraction of it.

This is to be compared with other methods for reducing the sequential task creation overhead:

- One method identifies points in the program in which none of the predecessors of a given codelet set has performed its puts. The prescriber codelets for the codelet set need to be run before the identified point. Beyond special cases such as wavefronts computations and collective one-way synchronizations between sets of codelets (for instance a big barrier between steps of an algorithm), identifying these points automatically - and without introducing additional synchronizations in the program - is a hard problem.
- Another method produces a decomposition of the codelets into a directed acyclic graph (DAG) of codelet sets, and computes both the dependences among codelet sets and within each codelet set[1]. On top of the additional synchronizations intrinsic to this method (*all* the codelet in a destination set transitively depend upon *all* the codelets in a source set), further synchronizations are required to ensure that all prescriber codelets are completed before the first predecessor of their prescribed codelet completes.

Finding optimal solutions for these two methods is a research question, and in particular, we don't know if there exist general solutions where the additional synchronizations are harmless. However, we expect that introducing additional atomic operations on low-contention objects will be much more efficient than introducing bulk dependences (and hence synchronizations) as in the other solutions.

Reservoir: Improving scalability of polyhedral core engine

R-Stream relies on a geometric internal representation of loop programs and accessed data as polyhedral domains. The vast majority of the loop parallelization and data locality optimization is performed in polyhedral representation, in which polyhedrons (and implicitly, the program they represent) are analyzed and transformed using R-Stream's polyhedral engine, "Jolylib."
When originally created, Jolylib was based on the ideas and principles of the then state-of the-art polyhedral library Polylib[2].

[1] On hierarchical architectures, this type of decomposition would be done hierarchically.
[2] http://icps.u-strasbg.fr/polylib/

In Polylib, a double representation is maintained for polyhedrons:

- as the conjunction of a set of constraints (equalities and inequalities), and
- as a sum of (the convex hull of) vertices, rays and lines.

Polyhedrons have a combinatorial structure: the vertices, rays and lines of a polyhedron can be obtained by combining (intersecting) subspaces defined by the constraints. Reciprocally, the constraints of a polyhedron can be obtained by combining its vertices, rays and lines. The number of vertices, rays and lines that can be obtained is bounded by the number of possible combinations of constraints, and conversely. As a result a relatively small number of constraints can translate to a very large number of vertices, rays and lines (and conversely). In the worst case, this number is exponential in the number of dimensions of the polyhedron. Hypercubes are a common illustration of this principle, as their number of inequalities (faces) is 2n (where n is the hypercube's dimension) and their number of vertices is 2^n.

The fastest known algorithm for computing vertices, rays and lines from constraints and back[3] is at the heart of the double representation in Polylib. While it is indeed fast for relatively small number of dimensions[4], its complexity is conditioned by the sheer number of vertices, rays and lines (or conversely, constraints) that need to be computed. Maintaining both representations hence inevitably results in computational costs that have exponential complexity in the number of dimensions of the manipulated polyhedrons and limits the tractability of polyhedral algorithms.

Recent work on polyhedral algorithms for parallelization[5] exposed that the polyhedrons used to represent loop programs are typically characterized by a relatively small number of constraints, the exponential growth residing in their vertices, rays and lines. Following this, newer polyhedral libraries obtain better tractability by only maintaining the constraint-set representation of polyhedrons. As a result however, some operations that were trivially formulated using vertices, rays and lines (e.g., the convex hull of a set of polyhedrons), are less obvious to formulate using constraints only. But in most cases, the computation of the entire set of vertices, rays and lines of a polyhedron is unnecessary. When only one of these needs to be found, linear programming (LP) solvers can often be used advantageously, as they offer low asymptotic complexity. However, the overhead of calling an LP solver often makes them impractical in low-dimensional cases, in which the dual-representation approach performs well. Nonetheless, it remains that the dual representation is fast for low-dimension, low-combinatoriality problems.

Considering these aspects, our solution is to allow Jolylib to manipulate both single- and double-representation of polyhedrons, and to adapt both the representation and the algorithms to the characteristics of the polyhedron. We have been building this principle into our new version of Jolylib, which is planned for a release mid-October 2013.

[3] N. V. Chernikova, "Algorithm for finding a general formula for the non-negative solutions of a system of linear inequalities", *Zh. Vychisl. Mat. Mat. Fiz.*, **5**:2 (1965), 334–337.

[4] Empirically "up to a dozen", although it really depends upon the polyhedron's structure.

[5] Sven Verdoolaege, "ISL: An integer set library for the polyhedral model," International Congress on Mathematical Software (ICMS 2010), Kobe, Japan, 13-17 September 2010, Lecture Notes in Computer Science, volume 6327, pages 299-302, Springer.

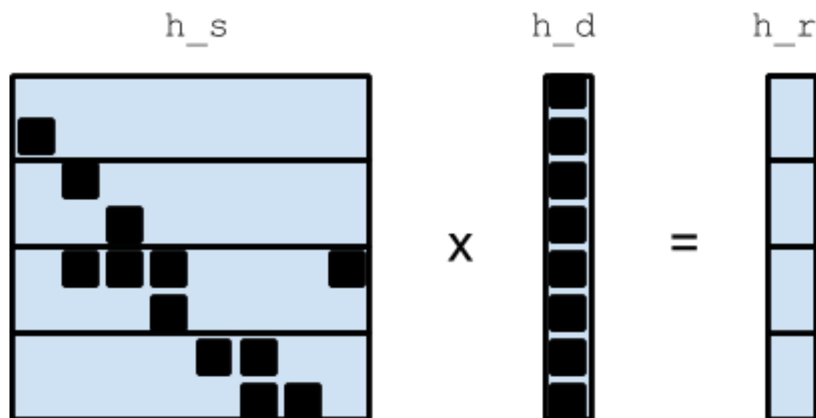
UIUC: Sparse Data Computation Library on PIL

In real world scientific computing applications, there are often times when the data we deal with is sparse in nature. Thus, it is very important to provide library routines for to efficiently store and manipulate sparse data in a parallel programming environment for application developers. During the last quarter, we developed a library to facilitate sparse data manipulation in PIL.

We based the library on our HTA class. The HTAs are dense by default. To add the support for sparse data with minimal modification, we made the necessary changes in function interfaces to specify the creation and initialization of sparse arrays. The functions that do not access scalar level data elements stay unmodified, since they work with the metadata describing the arrays instead of the raw data in sparse tiles.

The current version represents sparse tiles in Compressed Sparse Row (CSR) format. It is possible to extend the current library to add supports for other sparse data format with slight modifications. We chose to start with CSR because it is a common format which enables the use of well-known commercial or open source libraries later.

The following figure and code snippet demonstrate how to use sparse arrays in a PIL program. There are two input arrays. One is `h_s` (line 15), a sparse 2D NxM matrix partitioned to four tiles. The other is `h_d` (line 16), a dense 1D vector of size M. The results are stored into another dense 1D vector of size N, `h_r` (line 17), partitioned to four tiles.



```
01  // Include headers and constant definitions
02
03
04  void sdmv(int *target_id, int i, HTA h_r, HTA h_s, HTA h_d)
05  {
06    HTA s_tile = h_s->tiles[i-1];
07    HTA r_tile = h_r->tiles[i-1];
08    H3_SDMV(r_tile, s_tile, h_d);
09    *target_id = 0;
10  }
11
12  void pil_main()
13  {
14    // Allocate and initialize input and output arrays
15    // h_s is a sparse array of tiling (4, 1)(N/4, M)
16    // h_d is a dense array of tiling (M, 1)
17    // h_r is a dense array of tiling (4, 1)(N/4, 1)
18
19    pil_enter(SDMV, &h_r, &h_s, &h_d);
20
21    // At this point, the results are computed and stored in h_r
22    // . . . . .
23  }
24  node(SDMV, i, [1:4:1], target_id, [0], sdmv(&target_id, i, h_r, h_s, h_d))
```

The sparse matrix to dense vector multiplication is performed by calling `pil_enter()` (line 19) and specifying the PIL node (`SDMV`) to execute (line 24). A PIL node is, in essence, a parallel for loop, and the third parameter specifies its iteration space. In this case, four execution instances are created and each of them executes function `sdmv()` with a different `i` value. Inside `sdmv()` function body, we use `i-1` to get the correct pointer to the lower level tiles to work on, and pass them to `H3_SDMV()`, which is a serial sparse matrix to dense vector multiplication function.

By using the library developed, we are able to run CG benchmark, which is a conjugate gradient computation application of NAS Parallel Benchmark (NPB) specification, in both OpenMP and SCALE environment.

The above implementation is done in the Parallel Intermediate Language (PIL). PIL is capable of generating code for several backends, including the SWARM Codelet Association LanguagE (SCALE). For SCALE, each PIL node is implemented as three Codelets as seen in the figure below. There is one each of an Entry, Body, and Exit Codelet. Since PIL nodes are parallel, there can be multiple instances of the Body Codelet, as described in the user's program.