Critical Technology Evaluations

Technology	Description	Status
Parallel Language	Targeting distributed SCALE from PIL	Complete
Parallel Language	Sparse data representation in PIL	Identified
Multi-Node Parallelization	Parallelization of “twoel” section of optimized SCF code with MPI and SWARM multi-node	Evaluating
Multi-Node Parallelization	Parallelization of “diagon” section of optimized SCF code with MPI and SWARM multi-node	Evaluating
Kernel Comparison	Comparison of parallelized SCF kernel to the original global arrays version in NWChem	Complete
Compiler	Automatic single-node parallelization from mappable C to SWARM	Close to Completion
Compiler	Dependence computation simplification	Completed
Compiler	Simplified polyhedral domain computations	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	Identified
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
Application Development	Coupled cluster computation using the Tensor Contraction Engine identified as next NWChem co-design app. Working on C version of tensors.	Evaluating

Summaries of Quarterly Work (Q3)

UIUC Work

The major achievement in this quarter is the extension of PIL to target distributed SCALE. With the new extension, PIL can be programmed in a Single Program Multiple Data (SPMD) fashion. Once an executable is invoked, all Processor Elements (PEs) execute the same program concurrently. Each execution instance running on a PE manages local memory space and executes independently. Once in need of communication with each other, PIL invokes SWARM network APIs to send data to and receive data from remote PEs. A barrier API was also implemented for the explicit synchronization across PEs as intended by programmers. The PIL API document was updated to version v0.4. It reflects the recent changes to enable SPMD PIL programming. The new capabilities are illustrated with examples. This document can be found in the deliverables section of the DynAX XstackWiki page: https://www.xstackwiki.com/index.php/DynAX#Deliverables.

ETI Work

ETI has focused this quarter on further parallelization of the SCF benchmark provided by PNNL. Several potential optimizations were evaluated for increasing the parallelism of SCF beyond the improvements to the "twoel" module last quarter. Descriptions and results for all of these optimizations are detailed in the Topic Detail below.

PNNL Work

Power Efficient Data Abstraction Layer (PEDAL)

We are building the underlying infrastructure to guide our design of the Power Efficient Data Abstraction Layer (PEDAL) for Rescinded Primitive Data Type Access (RPDTA). PEDAL is being leveraged by both this project and the Traleika Glacier project.

In order to understand the interactions between applications and runtime systems we need measurement tools and assessment simulators/models to provide insight into data access patterns and lifetime, memory composition and structure residencies. We are developing both developing both a top-down and a bottom-up measurement framework.

For the top-down approach we are using Valgrind. This quarter we completed adapting Valgrind to SWARM. The framework measures lifetime in instruction counts, data address locations, access pattern and data content.

```
==19196== DHAT @awmm mod: WARNING: data output only valid for 64 bit words (e.g., double)!, a dynamic
heap analysis tool
==19196== NOTE: This is an Experimental-Class Valgrind Tool
==19196== Copyright (C) 2010-2012, and GNU GPL'd, by Mozilla Inc
==19196== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==19196== Command: ./cholesky D 50 10 5
**19196** Matrix address space is 79be540-7ba69c0
**19196** Filtering address space 79be540-7ba69c0
==19196== S First Codelet started
==19196== [cid, tid, alloc]   {I, J, Level, State}   (start_inst-end_inst)  Type    @start_addr-end_addr
outliers:reads/writes  step bytes
```

```
==19196== [3, 0, 1864331]     {-1, -1, 0, -1}              (14428016-15222031)    RNS*H   @79be540-
79c7228 330:389 8  3112
==19196== [3, 0, 1864331]     {-1, -1, 0, -1}              (15211476-15222038)    W*S*H   @79be540-
79c7228 38:87   8  696
```

Above is a small excerpt of the tool's output for a Cholesky SWARM run, aggregating Valgrind's lifetime, address, and pattern information with application information. A similar output (not shown), adds data content as well. We are currently developing post processing tools in order to analyze the information and feed it into high level machine models to assess concurrency, lifetime, and residency. At the current stage we are extracting a dependency graph to perform various types of analysis.

Next quarter, we will continue working on the post processing tools and provide concurrency, lifetime information under various machine model assumptions. We will also check for potential Valgrind cache integration paths.

Application Development

We are developing two NWChem modules for testing and evaluation. This quarter we began preparing the second module --- the tensor routines of the Coupled Cluster Computation. The computation source code is generated automatically from a description file of the tensor equations. We are rewriting the source code to a more readable human form using a data parallel model. We are providing the description files so that compiler teams can write their own automatic code generators. Next quarter, we will perform a complete code rewrite for the Coupled Cluster Computation, provide driver and input/output files, and consult with compiler teams on automatic code generators.

R-Stream compiler work (Reservoir)

We extended R-Stream to support the production of parrallel code on a multi-core x86 machine (i.e., a "single node"). The main aspects of this extension are that SWARM features such as parallel dependence management, asynchronous gets and macro-based codelet declarations are supported. We explain how so in Topic detail "Automatic single-node parallelization from mappable C to SWARM.

We also worked on keeping compilation time within desirable boundaries, by identifying the main bottleneck occurring when targeting codelet-based programs as the computation of dependence polyhedrons, and by finding ways of significantly reducing their computational complexity. Methods for doing so are presented in Topic detail "Dependence computation simplification."

Finally, we also started reducing the practical computational complexity of polyhedral operations in our polyhedral engine *Jolylib*, by making the computation of dual representation of polyhedrons optional. This a step toward the general improvement of compilation time in R-Stream.

Topic Detail:

ETI: Optimization and parallelization of SCF module

Further optimizations of twoel

Dynamic workload balancing in SWARM multi-node version

The **twoel** function is an N^4 series of nested loops, with iterator variables *i*, *j*, *k*, and *l*. The first multi-threaded version we wrote (in Q2) would fetch an *i* value using an atomic operation, and then do all of the *j*, *k*, and *l* work for that *i* value. The *i* values do not all represent equal amounts of work leading to load balancing issues. The problem is lessened by fetching smaller chunks of work at a time. We adjusted the atomic op to return a smaller subset of [*i,j*] values to work on, effectively removing the problem of workload imbalance between threads.

The multi-node version assigns *i* values to compute nodes in a round-robin order causing a similar workload imbalance problem; the balance between threads on a node are fairly well aligned, but there is no mechanism to line up the threads on a node with the threads on another node. Inter-node work imbalance affects the global sum reduction at the end of twoel, as one slow node can delay the whole reduction process from starting. The effect becomes more pronounced as the node count increases; on 128 nodes, we estimate that about 50% of the **twoel** time is spent idle because of inter-node work imbalance.

The problem is made worse by network effects skewing the **twoel** start times. At the end of the serial section (**diagon** and **dens**), the density matrix (g_dens) is broadcast to all compute nodes, this serves as input to **twoel** in the next iteration. The broadcast of g_dens takes a non-zero amount of time, and the data arrives on some nodes earlier than others. Nodes can start as soon as they receive the data, but higher numbered compute nodes start later (on average), which means that they finish later (on average).

The above issues make a clear case for balancing the workload between nodes better.

To achieve inter-node workload balance, we added some network code to request work units from node 0. Work is rationed via an atomic op of a progress index on node 0, and the result is sent back to the compute node which requested it. There is a round-trip delay between issuing the request and getting the response. Workers prefetch work requests to hide this delay, such that a new request is sent immediately before starting to compute on the work request unit the worker had received previously.

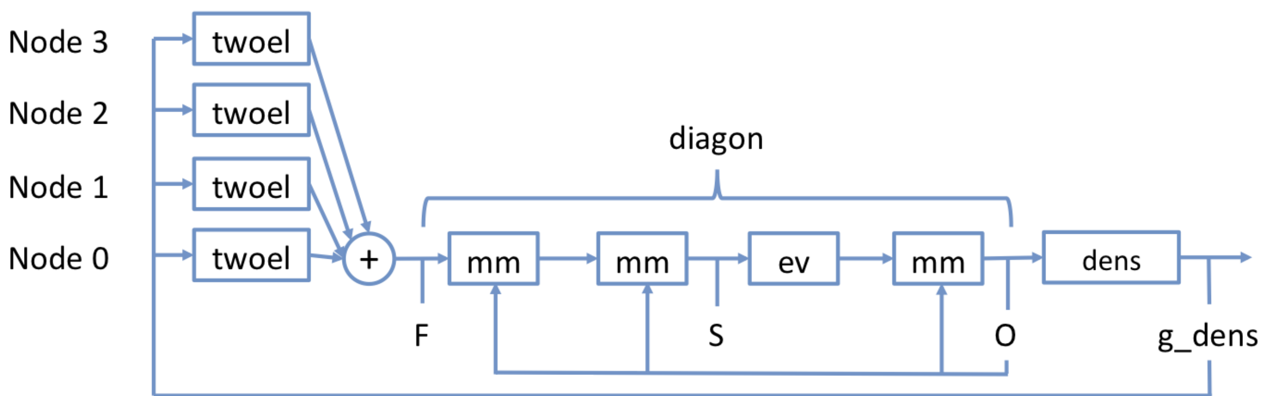
The work chunk size must be large enough to cover the network round-trip delay, yet small enough to minimize the effects of thread workload imbalance. We experimented with many different chunk sizes, but were unable to find a good balance between these two extremes. The work associated with an [*i,j*] chunk varies so much that increasing the chunk size enough to cover the round-trip delay produced considerable workload imbalance. In the end, we were unable to find any chunk size that performed better than the previous version of the code.

In execution traces, we also saw that node 0 started twoel earlier than any other node started, and also finished earlier than any other node finished. This is because node 0 has lower

latency access to the work allocation machinery, so it was able to start processing the requested work earlier, and on average, finish the requested work earlier. When the work allocation machinery runs out of work, node 0 becomes idle first, and the other nodes become idle some time later: that time representing the latency of the network transit and associated processing. So, variable access latency of the work allocation machinery is another source of work imbalance.

The current implementation represents a hub-and-spokes work topology, where node 0 (the hub) has the work and everyone else steals pieces of it. More complex work stealing topologies may improve this. For instance, each node could maintain a local cache of work for the threads on that node, and steal more from node 0 as necessary. Or the nodes could be arranged in a tree, where the amount of work stolen decreases for each level of the tree.

The figure below shows the parallelization at this point.

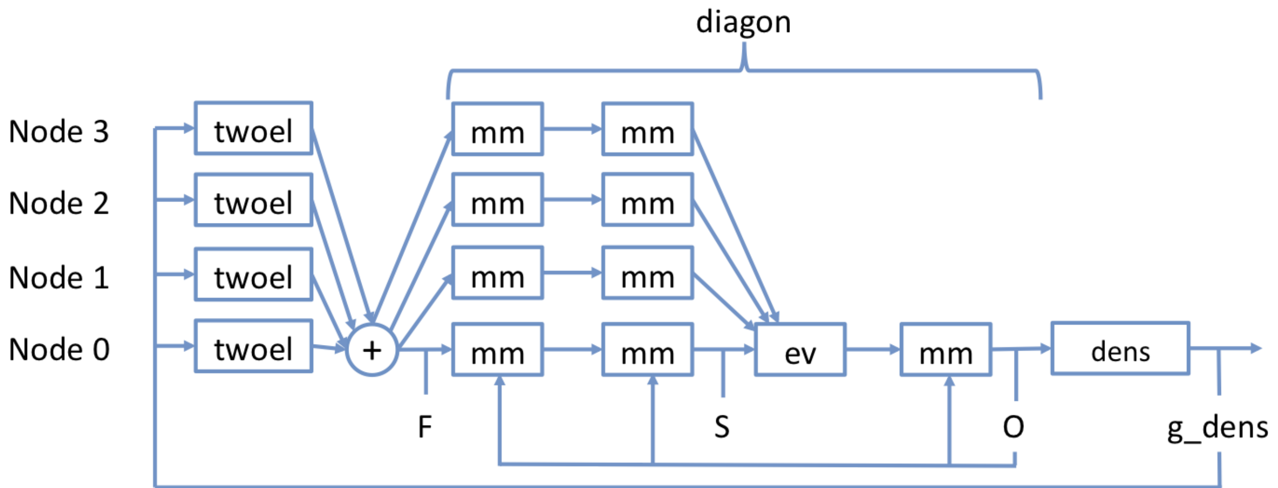


Optimizations of diagon

As mentioned in the Q2 report, the **twoel** step is embarrassingly parallel. **Twoel** keeps speeding up as you add more nodes; eventually the **diagon** step becomes the largest component of the overall execution time. During our testing, this happened at around 100 compute nodes in the SWARM multi-node version. **Diagon** uses multiple threads, because it uses concurrent MKL versions of lapack/blas functions, but it still only runs on a single compute node. It is desirable to spread the **diagon** work across nodes if possible, to bring the overall execution time down further. What follows are several steps we took while analyzing this problem.

Distribution of dsymm/dgemm

In this section we describe the parallelization of the two matrix multiplications after the twoel step. The figure below illustrates the parallelism achieved from the work described in this section.



Diagon starts by doing a similarity transform on g_fock, using g_orbs as the change of basis matrix. In other words, it calculates the following:

$$S = O^{-1} * F * O = O^{T} * F * O$$

Where F is the g_fock matrix, O is the g_orbs matrix, and S is a new matrix which is similar to g_fock. Note that because O is orthogonal, $O^{T} = O^{-1}$.

In the actual code, this becomes a series of two BLAS calls, using a temporary matrix T to store the intermediate result:

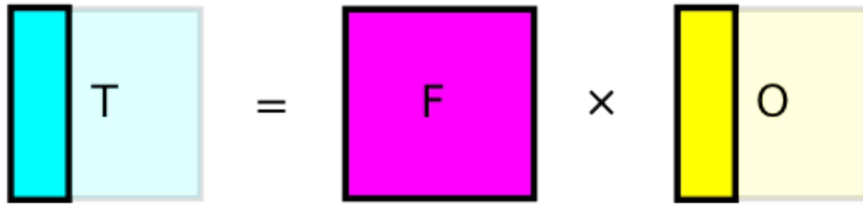
$T = F * O$ (using dsymm, since g_fock is symmetric)
$S = O^{T} * T$ (using dgemm, transposing g_orbs)

The old g_fock contents are no longer necessary, so the existing matrix is reused to hold the new result (S).

These operations can be distributed by having each compute node compute a stripe of the output fock matrix. Each compute node does require a copy of the full input matrices. However, the second matrix multiply can be computed solely with the T data generated from the first matrix multiply on the same node.

The first operation can be distributed by having each node compute a vertical slice of the (temporary) T matrix:

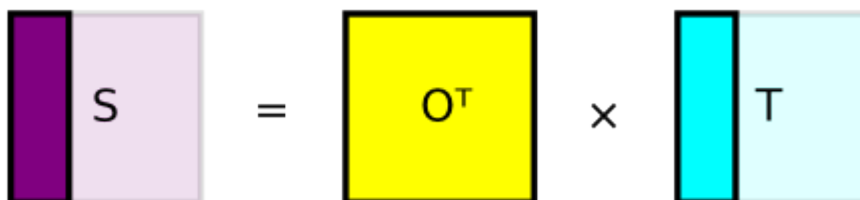
$$T[:,i] = F * O[:,i]$$



Where *i* is the range of columns assigned to the node.

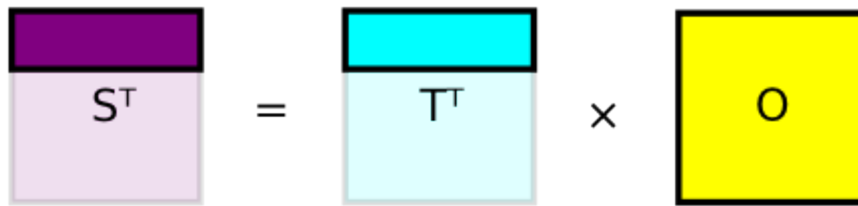
Then, multiplying that slice by the full-sized O results in a fully computed slice of the new F:

$S[:,i] = O^T * T[:,i]$



To efficiently gather the g_fock data at the end, it is advantageous for the slice to be contiguous in memory, representing the slice *i* of matrix S column-major order. In other words, from the perspective of the C language (where two-dimensional arrays are in row-major order), the result should be written into rows of the output matrix rather than columns. To that end, we use properties of transposition and perform a different operation that gives us the same result, in the format we want:

$S^T[i,:] = T^T[i,:] * O$



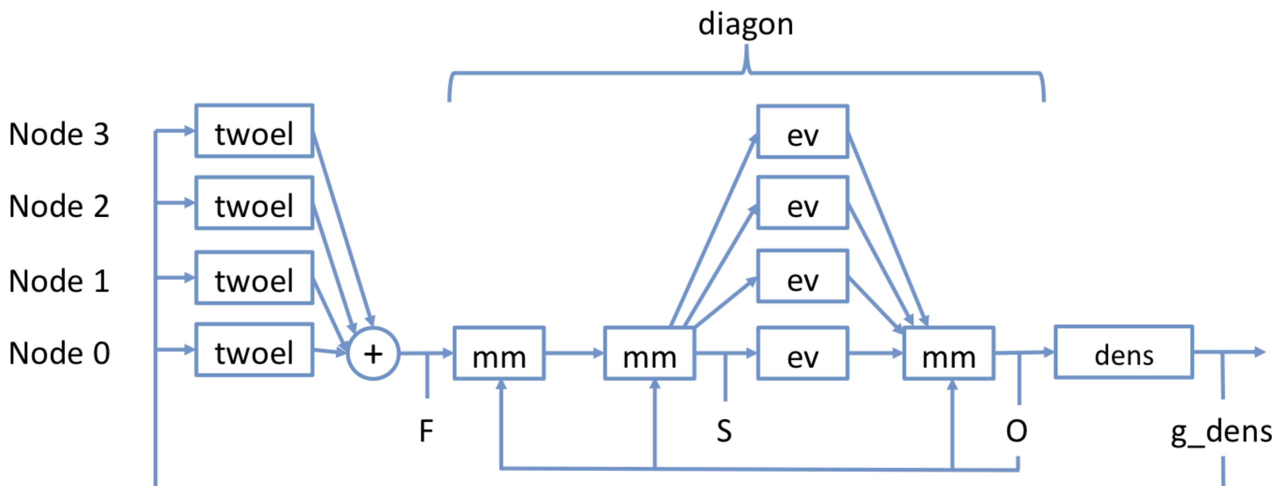
Where *i* is the same range as before, but now represents rows, not columns, of the matrix.

Now that the slice is contiguous in memory, it is easy for the network code to gather the data afterwards, simply copying each packet into the right location. The matrix multiply operations are implemented with dgemm, though each node's operation is *n* times smaller than the full single-node version, where *n* is the number of nodes.

In practice, the distributed matrix multiplications requires additional network traffic to set up. Specifically, the g_fock and g_orbs matrices need to be broadcast to all nodes. The latency of distributing g_orbs can be hidden because it is calculated before twoel, and not needed until the **diagon** step of the next iteration. The latency of distributing g_fock is critical, because it must occur directly after **twoel** reduces the matrix and before **diagon** begins. The net result was that the matrix multiply operations themselves were faster, but the overhead of the additional network traffic prevented an overall increase in measured performance.

Distribution of dsyev using scalapack

In this section we describe the parallelization of the eigenvector solver after the matrix multiplications. The figure below illustrates the parallelism achieved from the work described in this section.



The most costly part of **diagon** is the call to the blas function “dsyev”, which finds eigenvectors and eigenvalues of a symmetric matrix. The optimized MKL version of this function makes use of multiple threads, but is limited to running on a single node. **dsyev** represents about 70% of the overall **diagon** time, making it an obvious candidate for distributing across nodes.

As a first step of learning how best to distribute the dsyev, we decided to try scalapack, which has an existing implementation of distributed dsyev, called “pdsyev”.

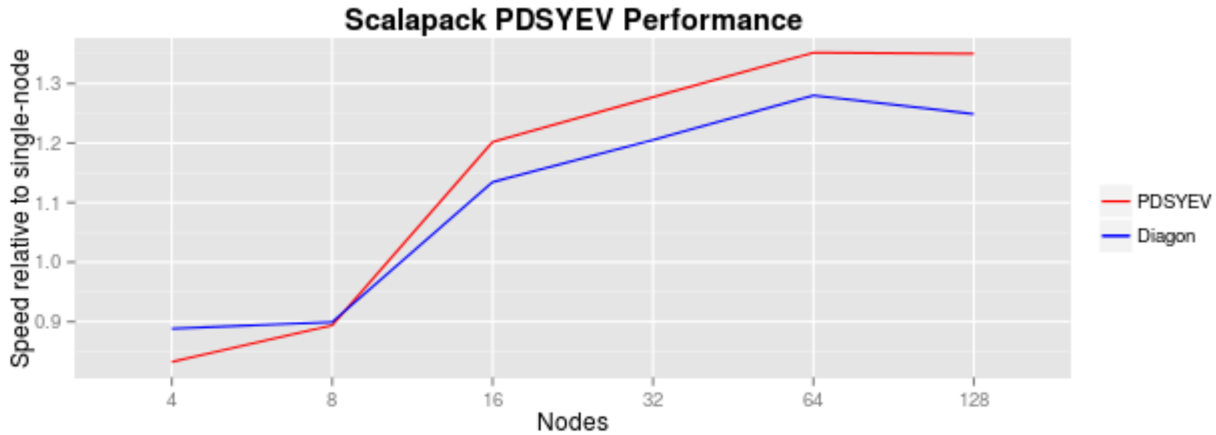
We adapted the multi-node SWARM version of SCF to be launched successfully by mpirun/mpiexec, and adapted it to initialize the MPI and BLACS environments properly. Since **diagon** is called from the main thread, it is possible to make MPI and scalapack calls within **diagon**. The program splits up the g_fock matrix into tiles, and defines a pattern that maps the compute nodes to tiles, using scalapack’s two dimensional block-cyclic distribution [see http://netlib.org/scalapack/slug/node75.html]. Each node allocates a temporary array to hold the tiles it considers “local”, the global data is copied into these local buffers, and then pdsyev is called to calculate the eigenvectors. Finally, the result data is copied back to node 0, to perform the last parts of the **diagon** work.

Unfortunately, when **diagon** is called from the main thread, it is unable to make use of SWARM networking and control flow constructs. Thus, this experiment was done independently of the dgemm distribution work mentioned above.

We saw the performance vary wildly with the parameters (tile size and pattern). It seems that larger tiles are always better, but of course, a larger tile size results in fewer total tiles. If the tile count is larger than the node count, not all of the nodes will take part in the operation.

The best numbers we saw were for a 375x375 tile size. The g_fock and g_orbs matrices are 3750x3750, for this input file, so there are 100 tiles in total, arranged in a 10x10 grid.

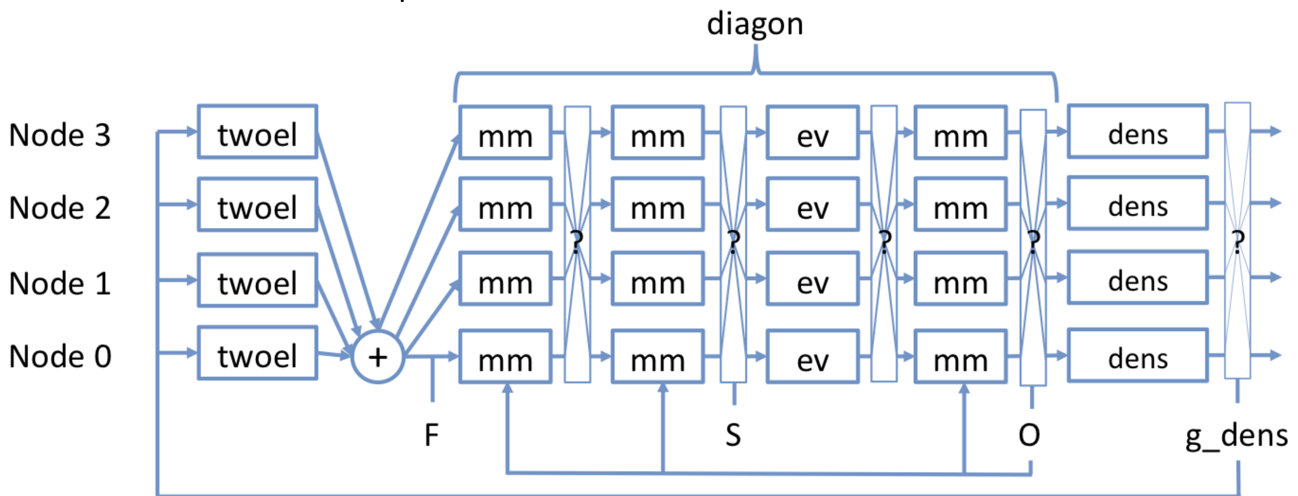
Node count	**Eigensolver**	**Total diagon**
(single node)	95.20	125.58
4	114.37	141.38
8	106.51	139.66
16	79.21	110.69
32	74.53	104.20
64	70.42	98.11
128	70.51	100.56



The above table and chart show the results of using scalapack to find the eigenvectors in **diagon**. The table contains the execution times in seconds; the chart shows the execution time relative to the previous (single-node) execution time. The "PDSYEV" line is relative to the standalone dsyev time; the "Diagon" line is relative to the standalone **diagon** time. Somewhere between 8 and 16 nodes, Scalapack reached the performance of the non-distributed version on a single node. After 16 nodes, performance continued to increase slowly. At 64 nodes, the performance improvement over a single node was 35% for the eigensolver itself, and 25% for **diagon** as a whole. After 64 nodes, this performance increase stopped. The lack of speedup between 64 and 128 nodes was probably due to the large tile size; some nodes had no local data to work on.

Problems and possible future work

In this section we describe future work on parallelization and data distribution for the entire **diagon** and **dens** computations. The figure below illustrates the parallelism we hope to achieve in future work. "?" indicates that the amount and type of data distribution is not yet known although the objective is to minimize the data movement and overlap computation with data movement as much as possible.



The matrix multiplication and the dsyev parallelization both suffered from overheads of data movement. There are two types of data movement we must consider. We define these as internal data movement and external data movement (to indicate where they sit relative to the

matrix operations). External data movement explicitly moves the data around such that it is in the right form, such as broadcasting g_fock or g_orbs ahead of time, or re-assembling the stripes of g_fock afterwards. Internal data movement is the communication which occurs during the process of the matrix operation, for instance, the scalapack matrix multiply may fetch copies of non-local tiles from matrices A and B to compute a tile of C. Both of these types of data movement can cause latency, which can lead to increases in execution time if the latency is not hidden with overlapping computation. The goal is to minimize the execution time of **diagon** as a whole.

In the first experiment, the data was well-placed for the matrix multiplications, with no internal data movement at all, but the supporting external data movement was costly. In the second experiment, the dsyev work was not distributed well, and thus the gains of extra CPU resources were mostly lost due to internal synchronization and data movement overheads. To use distributed computing power more effectively, we need to reduce the total overhead of the data movement. The extent to which this is possible remains to be evaluated.

In general, we want a data distribution that minimizes data movement while providing enough concurrency to keep the nodes busy. It is possible that a different set of parameters, which were not immediately apparent to us, would do a much better job of achieving those goals. Scalapack only provides an API to describe the high level data distribution, but gives us no indication as to the *right* distribution to use. Other than measuring wallclock time, we don't have any visibility on what happens internally, or how we can improve that process. More in-depth study is required to find a more optimal tiling pattern, both for the eigensolver itself, and for the other operations which surround it (tridiagonalization, orthogonal matrix generation, matrix multiplication).

The use of scalapack presents a second problem, which is that it follows a bulk synchronous schedule, with synchronization and data being sent at fixed points in the computation. It does not allow unrelated computations to overlap where CPU resources would otherwise be sitting idle, and it does not allow data movement to overlap with computation. This unnecessary synchronization hinders performance, and needs to be addressed. A tuned asynchronous solution would almost certainly be faster overall.

An ideal data layout would support the computation of each stage of **diagon**, with a minimum of internal data movement (communication inside each stage) and external data movement (data shuffling between steps). An ideal control flow would minimize synchronization, and allow the necessary data movement to overlap with computation as much as possible, to minimize the overhead and maximize the performance. Improvements in both of these areas would benefit **diagon**.

The dsyev function is made up of three stages, implemented as three separate functions. These functions are:

* dsytrd: double precision symmetric tridiagonalize
* dorgtr: double precision generate orthogonal matrix from elemental reflectors
* dsteqr: double precision tridiagonal matrix eigensolver

The following table shows the total number of seconds (wall-clock time, not CPU time; average of three runs) spent in various parts of **diagon**, through all 17 iterations required to

converge for a particular input file. These numbers represent the amount of processing needed for each step, without the overhead of network data movement. As such, they are taken from a single-node **diagon**, with no scalapack and no striped matrix multiplications.

dsyev	**dsyev**	**dsyev**		
dsytrd	**dorgtr**	**dsteqr**	**(other)**	**total**
30.08	24.51	53.03	44.77	152.39
28.78	24.46	51.77	46.14	151.15
29.18	25.26	53.47	43.99	151.90

The "other" column of the above table includes 3 matrix multiplications (the similarity transformation described above, and one other multiply), an n^2 search for the maximum absolute value of off-diagonal elements, and other assorted minutiae.

Each of these steps may perform differently for any given data tiling pattern. Future work on this problem should take this into account, and work toward a tiling pattern or a series of patterns which minimize the overall data movement overhead.

Reservoir: Automatic single-node parallelization from mappable C to SWARM

We developed support for generating parallel SWARM code from sequential C by designing and implementing a SWARM back-end and pretty-printer to Reservoir's R-Stream compiler. We started from the generic codelet generation engine (developed in Q1 and Q2) in the R-Stream polyhedral mapper, which uses the OCR back-end. We extended it to support the following SWARM features.

Parallel dependence management

In SWARM, each codelet defines its own dependences with other codelets in parallel using tag tables. Due to dependence declaration constraints in OCR, the R-Stream OCR backend uses centralized, sequential dependence declaration, which translates into a sequential cost of $O(V^2)$ for starting a parallel code with V codelets.

By opposition, with the SWARM back-end, this cost is reduced to O(V). This cost can even be further reduced through further optimizations, which will distribute the scheduling of tasks.
This difference in sequential startup cost may not be significant when V is small -- such as the single-node parallel programs we are producing as a first step -- but will be critical at Exascale where V is large.

Support for parallel dependence definitions required adapting the polyhedral mapper, as a dependence between codelets A and B translates to a "put" operation in A and a matching "get" operation in B (two matching "dependence operations").

Asynchronous gets

SWARM enables asynchronous “get” operations by associating a “get” operation with a re-scheduling of the codelet calling the “get.” This requires support in the generated code, which has to yield control when a get is called. The advantage of supporting this SWARM feature is that no codelet is waiting for a tag to become available and the processors can be fully utilized.

Macro-based codelet declarations

SWARM uses macros to hide the internal complexity of declaring codelets from the user (in our case, from the parallelizing compiler). Declaring a codelet boils down to declaring a “codelet descriptor” (which carries meta-data about the codelet) and specify the code for its corresponding codelet.

Producing macros is often challenging for a compiler that relies on a type-safe system since macros are not always isomorphic to function calls. Also, some of the SWARM macros entail syntax elements that diverge from standard C statements. This is mostly the case for the macros defining codelet (descriptors), as they look just like a function definition except that they don’t have opening and closing brackets, and that their parameters do not exactly match those of the underlying codelet function.

Support for this macro system was obtained by modifying R-Stream’s C pretty-printer to define exceptions for SWARM codes. Basically, SWARM codelets are marked when generated, and the behavior of the C pretty-printer is dependent upon the presence of markers.
To illustrate this, the figure below compares code for a pthread-based thread with code generated for a Swarm codelet.

```
/* Pthread code example */
void * function(void *); // forward declaration
void * function(void * THIS) {
    // code
}
/* Swarm code example */
swarm_CodeletDesc_DECL_LOCAL(function);
swarm_CodeletDesc_IMPL_LOCAL_NOCANCEL(function, function, swarm_c_void, swarm_c_void)
    // code
swarm_CodeletDesc_IMPL_END;
```

Further work

The code currently generated is untuned and limited to parallelization across multiple cores on a single node. We plan to extend this to multiple nodes using SWARM’s *nw* interface. Also, we plan to consider different heuristics for scheduling and tile size selection that are more specific to codelet computations, as well as optimizations of the R-Stream runtime layer for SWARM.

Reservoir: Dependence computation simplification

Our experience with the polyhedral representation, used by R-Stream, is that it enables a clean, mathematical and intuitive formulation of program parallelization and optimization algorithms. These advantages come with the need for controlling the computational

complexity of these algorithms, which often have exponential complexity in the number of dimensions of the manipulated polyhedrons.

In programming models that require expressing the program as a notional graph of codelets (such as SWARM, CnC and OCR), a naive way of computing dependences between two codelets is to check whether they access common data that at least one of the references writes.

When parallelizing loop codes, loop iterations are usually partitioned into tasks (through tiling). Outer loops are formed which iterate across the tasks (which are then expressed as codelets). Hence, each codelet code has several occurrences defined by the values of the iterators of the outer loops (which we'll call "inter-task" loops).

The naive dependence test between two codelet instances consists of computing the accessed data of each codelet as a function of the value of its inter-task loop indices, and get the values of the inter-task loop indices for both codelets for which the data sets accessed by both codelets intersect[1].

Unfortunately, this method relies on building a high-dimensional set of polyhedrons and projecting some of the dimensions out, leaving only the inter-task dimensions for both codelets. Projection is a highly combinatorial operation, and the computation time of a projection on a high-dimensional, complex polyhedron can easily make compilation time undesirably high.

The computation of dependences (which are in fact dependence polyhedrons) can then easily become the bottleneck in automatic parallelization to a codelet-based programming model.
We have addressed this issue through various techniques, which can be summed up as reducing the complexity of the dependence polyhedron and delegating some dependence computations to run-time. These optimizations are as follows.

Using loop types to define dependences

As part of its representation of loops, R-Stream decorates loop dimensions with "types", which summarize high-level properties of the dependences carried by the loops:

- The "doall" type represents loops that don't carry any dependences.
- The "perm" type represents permutability with loops directly above or below (they are defined for a set of consecutive loop dimensions). This translates to dependence vectors lying in the non-negative orthant.
- The "reduction" type represents loops whose iterations are associative. We have not focused on this type for this work.
- The "seq" type represents loops that don't have any of the previous properties. In an execution model based on loop parallelism (such as OpenMP), the consequence is that the iterations of such loops are executed sequentially.

---

[1] M. Baskaran, N. Vydyanathan, U. Bondhugula, J. Ramanujam, A. Rountev and P. Sadayappan, "Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors,"
[1] in proceedings of PPoPP 2009, pages 219-228.

When computing dependences, the “doall” and “perm” types can be used straightforwardly to generate conservative dependences without needing to compute a high-dimensional polyhedron:

- There is no dependence between iterations of a doall loop.
- Since they all lie in the non-negative orthant, dependences carried by perm loops can be conservatively approximated by the canonical unit basis vectors along the perm loop dimensions.

R-Stream extracts fully permutable loops from a range of kernels including stencils and LU decomposition. A conservative set of dependences -- which often turns out to be exact -- can then be computed without projecting out high-dimensional polyhedrons.

This technique was combined with the run-time dependence testing presented in the following section, by extending the poly-cnc framework, developed in the Traleika Glacier project, into a runtime-agnostic runtime wrapper that supports both CnC and SWARM.

Conducting run-time dependence test

Instead of having the R-Stream compiler generate explicit dependence polyhedrons that define which inter-task iterations of codelet B depend upon which inter-task iterations of codelet A, we can rely on knowing that loops are permutable to compute whether a “get” from or a “put” to a neighboring codelet instance is required by testing whether the neighbor’s iteration domain is non-empty.

In the runtime-agnostic wrapper mentioned above, this is performed by generating a slight variant of the task’ loops (parameterized by the inter-task indices), which doesn’t execute the loop but tests whether neighboring loops defined by the canonical vectors of the positive orthant (as explained in the previous section) contains at least one iteration.

Preliminary results show that we can parallelize deeper stencils (as high as 3D+time+ two size parameters) within a desirable amount of compilation time using the combination of loop-type-based dependences and run-time dependence testing. We plan to evaluate the separate effects of both techniques in Q4.

Performance numbers relative to OpenMP are also encouraging, although mixed, as they go from 20X speedup to 50% slowdown, depending upon the kernel. Relevance of such a comparison is arguable since tile sizes that are good for bulk synchronous execution are not necessarily good for codelet-based execution, and conversely. Hence, we are expecting to learn from these results, run a less arguable comparison and present our findings in the next report. A report presenting the method in detail and the preliminary performance comparison results is available separately from this document.

Convexification of dependences stemming from uniformly-generated references

Groups of uniformly generated array references are characterized by access functions that only vary by a constant. For example, in the following loop nest, the three references to A are part of a uniformly generated reference group (UGR):

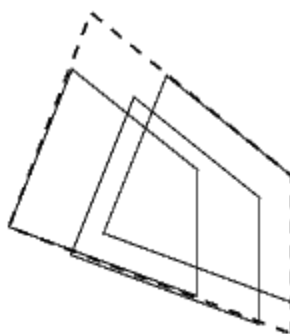
```
for (int i=0; i< N; i++) {
```

```
    for (int j=0; j<M; j++) {
        A[f(i,j)+1][g(i,j)] = A[f(i,j)][g(i,j)-1];
        …  = A[f(i,j)-3][g(i,j)]+B[i] ;
    }
}
```

Computing the exact dataset of A accessed by the example loop nest boils down to computing the union of the image of the iteration domain {0 =< i <N; 0 =< j <M} through each of the three access functions to A.

Unfortunately, the result is a non-convex set of three polyhedrons. Computing a dependence polyhedron using such an accessed dataset results in three times the number of projections as if there were only one convex dataset.

Instead, we can compute a simple, convex slight over-approximation of the set of accessed data by a polyhedral "inflation" technique. The method, graphically illustrated in the figure below , produces a data set (called "inflated hull") that is homomorphic to the iteration domain (which keeps its shape simple) and tightly includes the exact dataset accessed by the uniformly-generated reference group (the three references to A in our example).



Convex, inflated hull of three polyhedrons

Using over-approximations entails the risk of producing dependences from (or to) non-existent codelets. We avoid this risk by intersecting the over-approximated dependence polyhedrons with polyhedrons representing the set of actual source and destination codelets.

The larger the number of references in a UGR, the bigger the savings in computing the dependence polyhedrons. For instance, if both the number of UGR references in the source and destination codelets is 3, computation of the (one convex) dependence polyhedron for these references is about one order of magnitude faster.

Further work

We are constantly working to further improve the computation of dependences. We plan to implement a technique presented here, which is expected to reduce the number of dependence polyhedrons by the number of inter-task dimensions, and to produce significantly better dependence-management code.

Dependence convexification using runtime lexicographic test

When computing dependence polyhedrons between codelets A and B, we are forming either a read-after-write ("true"), write-after-read ("anti") or write-after-write ("output") dependence

depending upon whether A reads the commonly accessed data before B. Hence, dependence polyhedrons are defined by the conjunction of “A shares data D with B” with either “A accesses D before B” or “B accesses D before A.” Unfortunately, in the polyhedral model, relations of the type “A accesses D before B” are represented using non-convex sets of polyhedrons, which are intersected with the polyhedrons representing “A shares data D with B” to form the dependence polyhedrons.

Our method is to postpone the evaluation of whether “A accesses D before B” or “B accesses D before A” to a runtime comparison of the inter-task indices of the considered A and B codelets. As a function of this, codelet A produces either (respectively) a “put” or a “get” towards B. As a result, R-Stream only needs to compute the much simpler polyhedron “A shares data D with B.” A function, which will unify the declaration of puts and gets, will be added to the R-Stream runtime layer for SWARM to support this.