Table of Contents

Critical Technology Evaluations

Technology	Description	Status
Compiler	Automating the use of symmetry in input data (SCF)	Identified
Compiler	Automating pre-computation of pure function values (SCF)	Identified
Parallel Language	Completed first PIL API document	Complete
Parallel Language	SCALE 0.11 code generation	Complete
Parallel Language	Targeting distributed SCALE from PIL	Identified
Applications	Identified two NWChem kernels for study	Complete
Applications	Delivered C-only version of the Self-Consistent Field computation	Complete
Applications	Investigating Coupled Cluster method for second NWChem module	Evaluating
Serial Optimization	Optimization of C-Only SCF kernel. 15x-20x speedup	Complete
Single-Node Parallelization	Parallelization of optimized SCF code with OpenMP and, SWARM single-node	Complete
Multi-Node Parallelization	Parallelization of optimized SCF code with MPI and SWARM multi-node	Evaluating
Runtime Collective Design	Identified asynchronous broadcast and multi-part reduction in SWARM as possible speed improvements over MPI	Identified
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

Summaries of Quarterly Work

Deliverables

This quarter's deliverables include this report, an API document for the PIL library, the SCF module code, and SCF module optimizations and parallelizations. All of these can be found at https://www.xstackwiki.com/index.php/DynAX#Deliverables under the section: "Q2".

UIUC Work

This quarter we completed the first draft of the PIL v0.3 API document with its associated high level language, SPIL. This API includes a data structure for creating and manipulating tiled arrays, as well as fundamental parallel operations on tiled arrays. The arrays may be either dense or sparse, and dense arrays may be laid out either row-major or tile-major. Operations for expressing data associations with codelets as well as data movement operations are provided. There is a notation to express task graphs.

The high level language, SPIL, provides a language with syntactic sugar for common operations in PIL for programming convenience and program readability. The SPIL will have a source-to-source translator to generate code in PIL.

The PIL API document can be found at https://www.xstackwiki.com/index.php/DynAX#Deliverables

Work for PIL v0.2 generating SCALE was continued. A bug in the scheduler of SCALE 0.9 was identified. SCALE 0.11 was released to fix this bug and allow access to distributed SWARM. PIL v0.2 now targets SCALE 0.11 and work to target distributed SCALE is in progress.

ETI Work

ETI has focused this quarter on optimizing and parallelizing the SCF benchmark provided by PNNL. Code was parallelized on a single node using OpenMP and SWARM. Code was parallelized across nodes using MPI and SWARM. Results for all of these optimizations and parallelizations are detailed in the Topic Detail below. Serial improvements show a ~10X improvement over the base code. Parallel improvements show linear scaling on one node and scaling of the twoel procedure across nodes up to 256 nodes. We have released the documentation and code under BSD open source on the xstackwiki.com webpage under the "Deliverables" section of the Dynax page at https://www.xstackwiki.com/index.php/DynAX#Deliverables

PNNL Work

PNNL Task 6- This quarter we completed and delivered a simple, C-only version of the SCF module of NWChem. The release includes an input file, output file, and optimization suggestions. We prepared a powerpoint presentation and Word document describing the modules' mathematics, control flow, and data structures. With regards to the second module, the Coupled Cluster method, we have decided that the computational routines are not appropriate for our purposes; however, the routines are called from a driver that maneuvers a DAG of the data dependencies among the routines. We believe the driver plus the routines may serve as an appropriate module.

PNNL Task 9 - The task is structured following a 3-prong approach:
1) We are developing a fundamental theoretical framework we call “Group Locality” that will guide us towards data locality assessment shared by a set of threads.
2) We are performing a survey of compression algorithms suitable for scientific computing that will lead to a report in the next quarter. Based on the current state of the art, we will perform a down-selection that optimizes for low overhead and high compression ratios.
3) We are developing two measurement frameworks to help us understand the lifetimes of the underlying data structures in SWARM: One framework is based on Intel’s PIN instrumentation tool set. It is currently in the early debugging stage. The second framework is based on Valgrind. This framework is in a more advanced stage and already capable of performing some rudimentary analysis on the SWARM Cholesky code provided by ETI.

R-Stream compiler work (Reservoir)

This quarter we have focused on two main areas:

1. Optimizing abstract codelet generation. We designed a lightweight method for reducing transitive dependences generated by the dependence-declaring high-level operations created by the mapper. These operations translate to point-to-point synchronizations after code generation. This technique is close to completion. It will be presented in the next report.
2. Finding out how to automate the most impactful optimizations implemented by hand on the SCF kernel. Let n be the trip count of every considered loop. One optimiztion takes advantage of the symmetry in the values of a pure function (g) that is evaluatedn^4 times in SCF to reduce the range of evaluated values to $n^4/8$ using three symmetries. The other optimization targets subexpressions of g that are evaluated n^5 times but for which there are only n^2 distinct values. The goal is to precompute the n^2expression values and use the pre-computed values instead of re-computing them. The savings in terms of computation are in O(n^5) and the cost in memory is from 1 to n^2.

Topic Detail:

PNNL: Self-Consistent Field Method - A NWChem module for evaluation

PNNL has completed and delivered the first of two NWChem modules. The Self-Consistent Field Method (SCF) is often the central and most time consuming computation in ab initio quantum chemistry methods. It is used to solve the electronic Schrodinger Equation,

$$H\Psi = E\Psi.$$

SCF assumes that each particle of the system is subjected to the mean field created by all other particles resulting in a non-linear system solvable by an iterative, fixed-point algorithm.

By expressing the molecular system's one electron orbitals as the dot product of the system's eigenvectors and a set of Gaussian basis functions

$$\emptyset_i = \sum_i C_{iv}\, \chi_v\, (r)\ , \quad [1]$$

the solution to the Schrodinger Equation reduces to the following self-consistent eigenvalue problem

$$F_{\mu v} = h_{\mu v} + \frac{1}{2} \sum_{\omega\lambda} [(\mu v|\omega\lambda) - (\mu\omega|v\lambda)]\, D_{\omega\lambda} \quad [2]$$

$$F_{\mu v}\, C_{kv} = \varepsilon\, S_{\mu v} C_{kv} \quad [3]$$

$$D_{\mu v} = \sum_k C_{\mu k} C_{vk} \quad [4]$$

where ***F*** is the Fock matrix, ***C*** are the eigenvectors of the system, ***ε*** are the eigenvalues, ***S*** is the electron force overlap matrix, ***D*** is the system density matrix, ***h*** are the one-electron forces, and the terms in the square brackets in Equation 2 are the two-electron Coulomb forces and the two-electron Exchange forces, respectively.

Figure 1 depicts the method's control flow. The upper two leftmost modules initialize the ***D*** and ***C*** matrices allowing the first iteration to compute Equations 2 and 3. The *Construct Fock Matrix, Compute Orbitals,* and *Compute Density Matrix* modules compute Equations 2, 3, and 4, respectively. The *Damp Density Matrix* module scales the density matrix and finds the greatest changed value between the current and previous density matrix. If the absolute value of the change is less than a threshold value, the method terminates; otherwise, a new iteration is started. The method terminates after 30 iterations if convergence is not reached. Figure 2 gives the modules' names, and array inputs and outputs.

The modules comprise rectangular, nested for loops of the form

oneel

```
for (i = 0; i < nbfn; i++) {
for (j = 0; j < nbfn; j++) {
  g_fock[i][j] = (g_schwarz[i][j] > tol) ? h(i,j) : 0.0;
} }
```

makden

```
for (i = 0; i < nbfn; i++) {
for (j = 0; j < nbfn; j++) {
  double p = 0.0;
  for (k = 0; k < nocc; k++) p += g_orbs[i][k] * g_orbs[j][k];
  g_work[i][j] = 2.0 * p;
} }
```

dendif

```
for (i = 0; i < nbfn; i++) {
for (j = 0; j < nbfn; j++) {
  double xdiff = fabs(g_dens[i][j] – g_work[i][j]);
  if (xdiff > denmax) denmax = xdiff;
} }
```

where *nbfn* is the number of basis functions (= 15 * number of atoms). The loops are embarrassingly parallel, but the inclusion of reduction operations such as sum and max require concurrent atomic updates.

The most computationally intensive routine is **twoel** that computes the two electron forces,

twoel

```
for (l = 0; l < nbfn; l++) {
for (k = 0; k < nbfn; k++) {
for (j = 0; j < nbfn; j++) {
for (i = 0; i < nbfn; i++) {

  if (g_schwarz[i][j] * schwmax       ) < tol2e) {icut1 ++; continue;}
  if (g_schwarz[i][j] * g_schwarz[k][l]) < tol2e) {icut2 ++; continue;}

  icut3 ++;
  double gg = g(i, j, k, l);
  g_fock[i][j] += (     gg + g_dens[k][l]);
  g_fock[i][k] -= (0.5 * gg * g_dens[j][l]);

} } } }
```

The time for all the other routines is much smaller than the time for twoel. The guards in the innermost loop impose a cutoff limit and reduce significantly the number of updates computed. The guards can be hoisted to reduce loop overhead as explained in the optimization file that accompanies the code release. We note that the ***h*** values (one electron forces) and ***g*** values (two electron forces) used in computing the Fock matrix are constants, so they could be precomputed, saved, and reused. While ***h*** is $nbfn^2$, ***g*** is $nbfn^4$ making it impractical to save ***g*** for even small molecular systems. As explained in the optimization file, symmetries in ***g*** can be exploited to minimize storage and computation requirements.

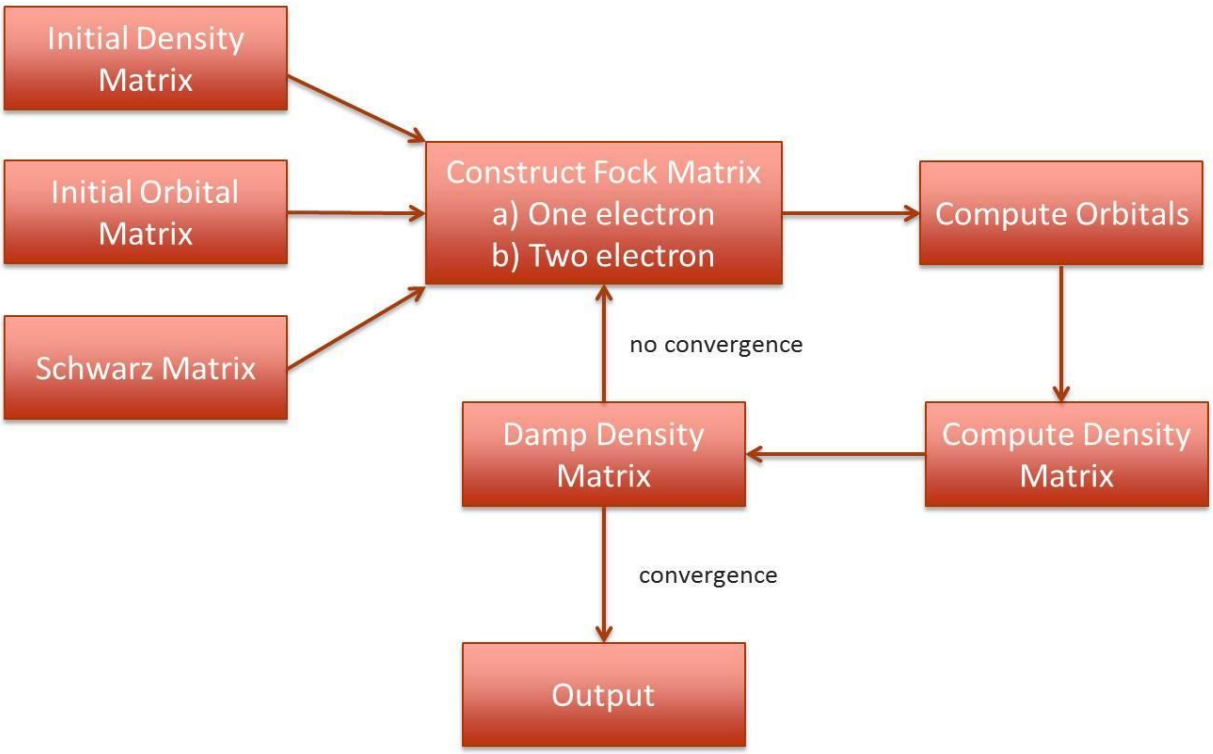


Figure 1 – SCF control flow

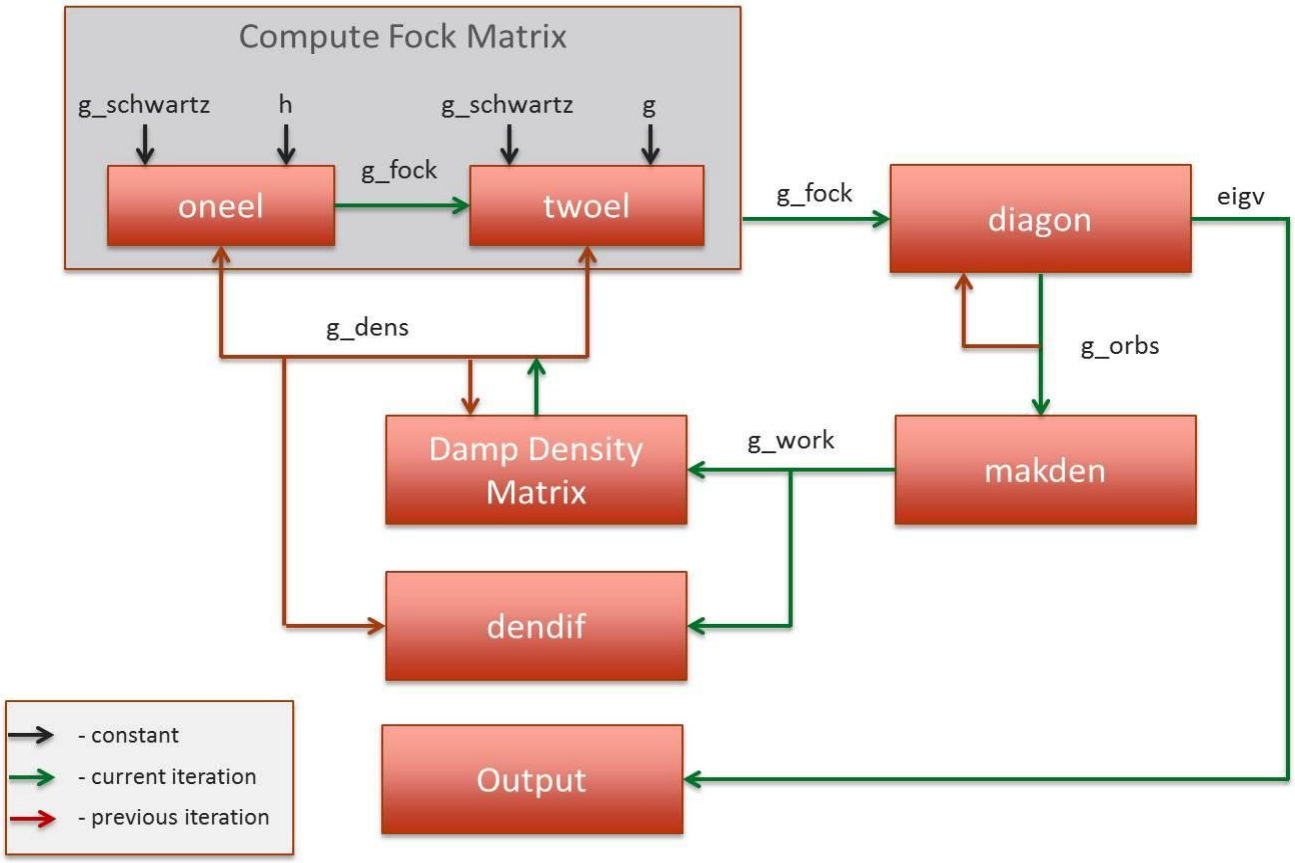


Figure 2 – SCF modules, inputs, and outputs

ETI: Optimization and parallelization of SCF module

Serial optimizations

Additional Sparsity Tests

In the reference SCF code, there were two sparsity tests in the **twoel** function's loops, one at the i,j level which allows skipping N_2 iterations and one at the i,j,k,l level which only skips a single iteration. We identified a third sparsity test that can be added to the i,j,k level which allows skipping N iterations.

Symmetry of g()

The **g** function called by **twoel** and **oneel** has three symmetries: I and J can be swapped, K and L and be swapped, and the (I,J) pair can be swapped with the (K,L) pair. The calculation and result is identical for any combination of these symmetries, leading to a total eight-way symmetry. To take advantage of this symmetry, **twoel** was modified to only ever call **g** with one of the possible symmetric inputs, then perform all the appropriate updates to the fock matrix.

This modification required decomposition of the N_4 twoel loop into six separate loops to properly handle the cases where the I,J,K,L arguments to g have restrictions which preclude utilizing the full symmetry. As an example, when I and J are equal, updates to the fock matrix cannot include the swapped version as it would not have been performed by the original twoel function.

1. Eight-way symmetry: IJ, KL and (I,J)(K,L); N_4 iterations
2. Four-way symmetry: I=J, KL and (I,J)(K,L); N_3 iterations.
3. Four-way symmetry: IJ, K=L and (I,J)(K,L); N_3 iterations.
4. Four-way symmetry: IJ, KL and (I,J)=(K,L); N_2 iterations.
5. Two-way symmetry: I=J, K=L and (I,J)(K,L); N_2 iterations.
6. No symmetry: I=J, K=L and (I,J)=(K,L) ; N iterations.

All together, this modification reduces the number of expensive **g** calculations by nearly eight while maintaining the same total number of updates to the fock matrix.

Lookup Tables in g()

The **g** function performs many computations which only rely on the (I,J) or (K,L) argument pairs. These calculations can be pre-computed and stored in a lookup table to reduce the number of required calculations. The lookup table requires $O(N^2)$ memory, which is the same order as the existing matrices used in the SCF code. In addition to reducing calculations, using a lookup table also reduces memory accesses as each value retrieved from the table generally would require two separate accesses to O(N) length arrays. Both pairs of arguments can use the same lookup table as swapping them was one of the symmetries in the **g** function.

Standard BLAS and LAPACK operations

When the input problem was increased to larger atom counts, the optimized **twoel** and **g** functions exhibited better scaling than several other operations. Specifically, the **diagon** phase of the computation can require more time to execute than **twoel** if the input problem is large enough.

To improve the performance of these operations, several BLAS and LAPACK functions can be used as drop-in replacements. The use of an optimized or parallel BLAS and LAPACK implementation drastically improves the speed of these operations and causes twoel to again dominate the total execution time.

Symmetry of fock and dens matrices

The number of memory operations performed in **twoel** can be approximately halved since the dens input to **twoel** is a symmetric matrix, and the resulting fock matrix will be symmetric. Because of the matrix symmetry, only one triangle of the matrix actually needs to be calculated, the other triangle can easily be reconstructed or may not even be necessary if a symmetric matrix multiply is used as the first action in **diagon**.

In order to implement this, the coulomb and exchange contributions to the two-electron forces must handled separately as they have different symmetries. The coulomb contribution has an additional symmetry which allows a further halving of updates by instead multiplying the contribution value by two. The exchange contribution is more difficult to deal with as its symmetry doesn't match that of g, and requires further loop decomposition to limit updates to only one triangle of the matrix.

We did not further decompose the loops to limit the exchange contribution updates to one triangle. Instead, we took some advantage of the symmetry to eliminate writing the same value to both symmetric locations. This resulted in a matrix which is not symmetric, but can easily be reconstructed into a symmetric matrix. The symmetric matrix is reconstructed by summing the values from both symmetric locations in the upper and lower triangles then writing that sum back to both locations.

This optimization was only added to the N^4 and N^3 decomposed **twoel** loops. Adding this change to the other loops would require additional decomposition, but wouldn't be likely to result in substantial performance improvements. Fully decomposing all the loops so that only one of the triangles is updated would also improve the CPU's cache utilization, as there would be fewer unique memory locations accessed by the **twoel** loops.

Cache g() results

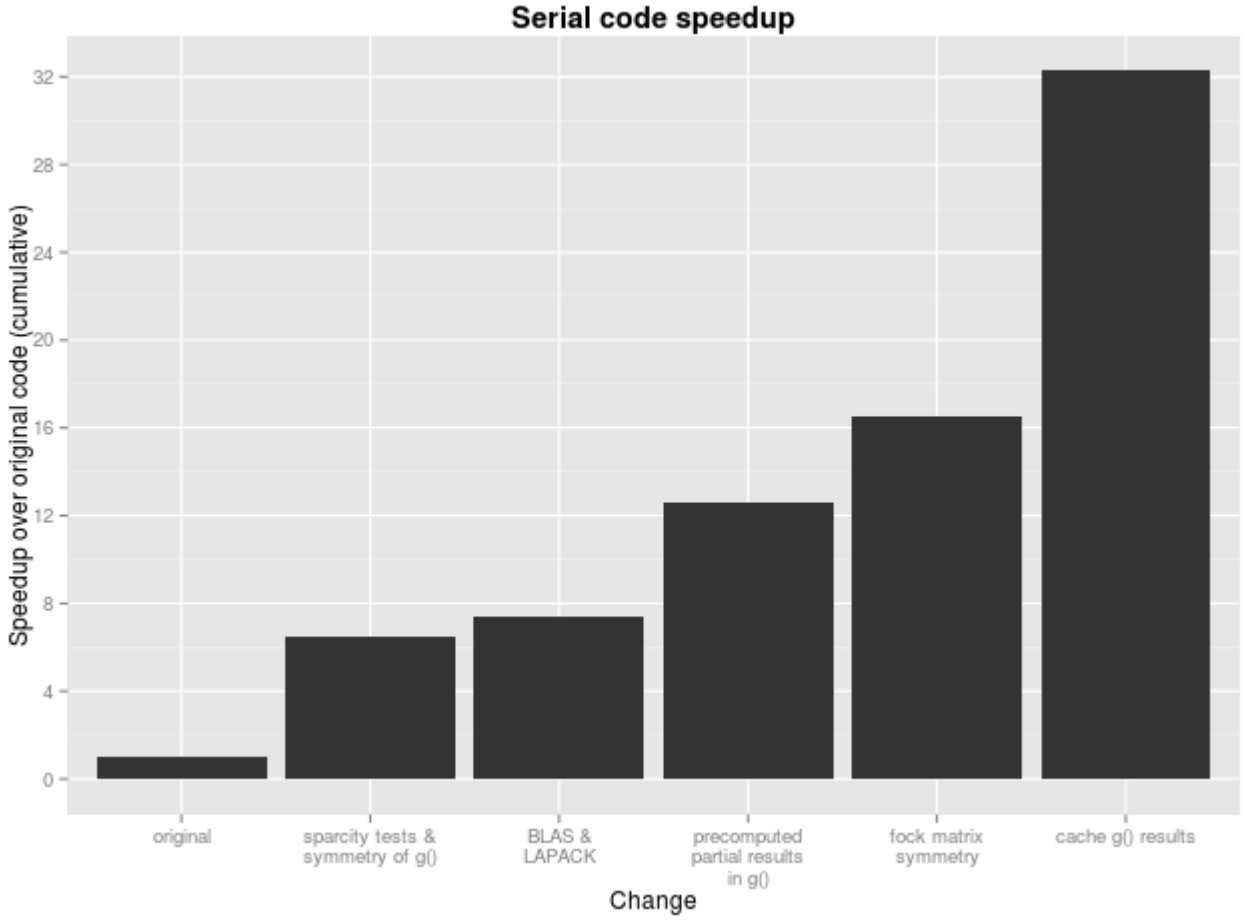
A very straightforward implementation of the conventional method was added which simply caches the results of the **g** function in the order in which it is called. Because the order in which **g** is called with various arguments does not change between iterations, we were able to append results to an array during the first iteration, then in subsequent iterations values are retrieved in the same order from the array. This method minimizes the memory requirements of caching results as only the required values are stored and there is almost no overhead required to map input arguments to the location of the cached result. The only overhead of this implementation is three additional index or pointer variables, one to store the end of the allocated space, one to store the end of the used space, and one to store the index into the array for values that are being either read or written.

Because this implementation of the conventional method requires that **g** is always called in the same order, it is not suitable for implementing as-is in the multi-threaded or multi-node versions of the SCF program since they would not have the same guarantee on the order in which **g** is called.

The memory requirements to cache results of **g** are not known until after the cache has been created and populated during the first iteration. At maximum the memory requirements would be on the order of N^4, but the actual requirements are typically much smaller, although still substantial, due to the sparsity increasing as the problem size increases. The incremental speedup of using the conventional method with all other optimizations compared to the direct method with all other optimizations is about 2x. Using the conventional method likely changes the limiting resource from CPU speed to memory bandwidth while also increasing power consumption.

Results

The following figure shows the speedup in overall program execution, for 50 atoms in a straight line.



Single-node Parallelization

OpenMP

Initial parallelization was performed for the **twoel**() function which performed up to $O(N^4)$ work in 6 different loop nests. Because OpenMP does not support array based reduction in the C standard, we show 3 possible ways to solve this problem.

1. Utilize the *atomic* directive on each update to the g_fock matrix. This would imply a locking mechanism per element with more exposed parallelism
2. Utilize the *critical section* directive in each section of updates in the loops. This would require as few as 1 locking operations per 8 updates but limit parallelism to one thread in the critical section at once
3. Allocate auxiliary space outside the loop nests, 1 per openmp thread, and perform a single threaded reduction at the end of all loop nests. This is the most similar to how an openmp reduction would be performed if the standard supported it.

Due to the variable nature of the calculations in loop enumerations pruned using Schwarz inequality, we looked at guided, dynamic, and static scheduling scheme provided by OpenMP.

SWARM

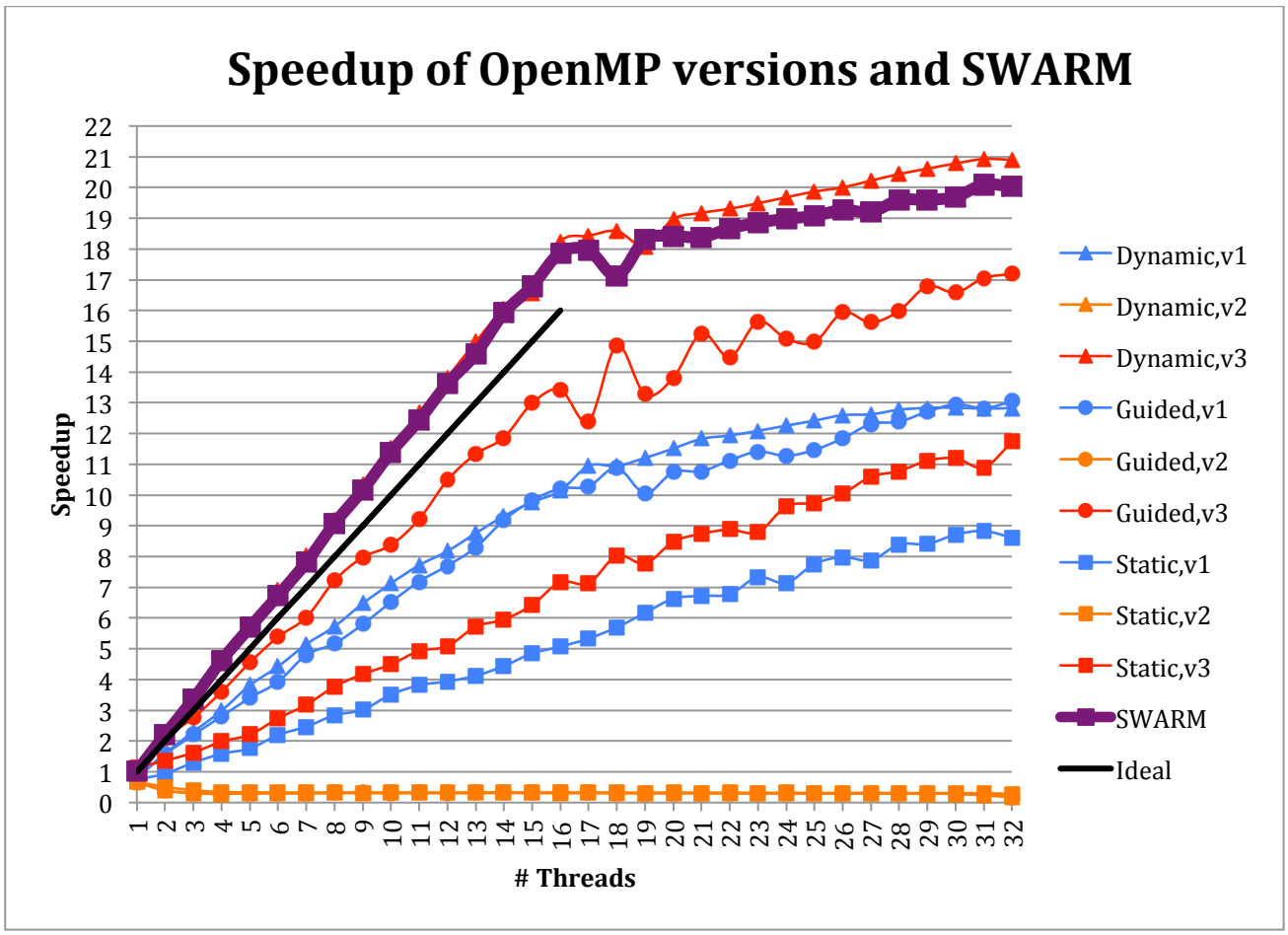
A single-node parallel version of the SCF program was created using SWARM in which only the **twoel** function was implemented in codelets, the remainder of the program is the same serial implementation as the the optimized serial version.

The **twoel** function was split into several codelets:

- Initial codelet: initializes reduction dependency, schedules a worker codelet for each thread on the system, then executes all N_2 and smaller **twoel** loops.
- Worker codelet: processes chunks of the two N_3 loops, then processes chunk of the N_4 loop. Each worker codelet running in separate threads has its own local partial fock matrix which updates are written to. Work units are selected by each thread performing atomic get and add operations to a shared global, the resulting value is then used in place of the outermost for loop.
- Reduction codelet: run after all other codelets have run and satisfied the dependency, accumulates all partial fock matrices into the main g_fock matrix

Results

The following chart shows the speedup of **twoel**, for 150 atoms in a straight line, over the same input running on a single thread. This is running on a machine with 16 cores, and 2 hyperthreads per core.



The higher-performing programs show a distinct elbow at 16 threads, this is when the number of CPU cores was reached. Hyperthreading allows performance to continue to increase after this, though at a much lower rate.

Another thing to note is that the SWARM and OpenMP Dynamic v3 programs seems to run slightly faster than expected on multiple threads, showing better than linear speedup on 2 through 17 threads. One possible reason for this would be fewer cache misses, when data has already been read by a nearby core.

The SWARM implementation is on par with the best OpenMP implementation until hyperthreading kicks in. At this point, OpenMP has a slight advantage. We will be looking into the difference in the next quarter.

Multi-node Parallelization

MPI

Assuming an OpenMP solution at the node level, we looked at using MPI to statically distribute work out. Very similar to the OpenMP solution, we assign different nodes to perform different iterations of the $O(N^4)$ twoel loop, and accumulate the g_fock matrix at the end of all loop iterations. Fortunately, the only inputs into this loop which varied between iterations of the outermost convergence loop was

g_dens, the electron density matrix, which was order $O(N^2)$. This created a simple MPI program which broadcasted a new density matrix, performed OpenMP optimized computations, and reduced the resulting g_fock matrix to a single node for processing. While it would be possible to send specific partitions of g_dens to only the nodes needing them and thereby possibly reducing the overall network bandwidth, broadcasting the entire density matrix to all nodes using a highly optimized MPI routine was a far more efficient technique and could allow for some level of load balancing with additional effort.

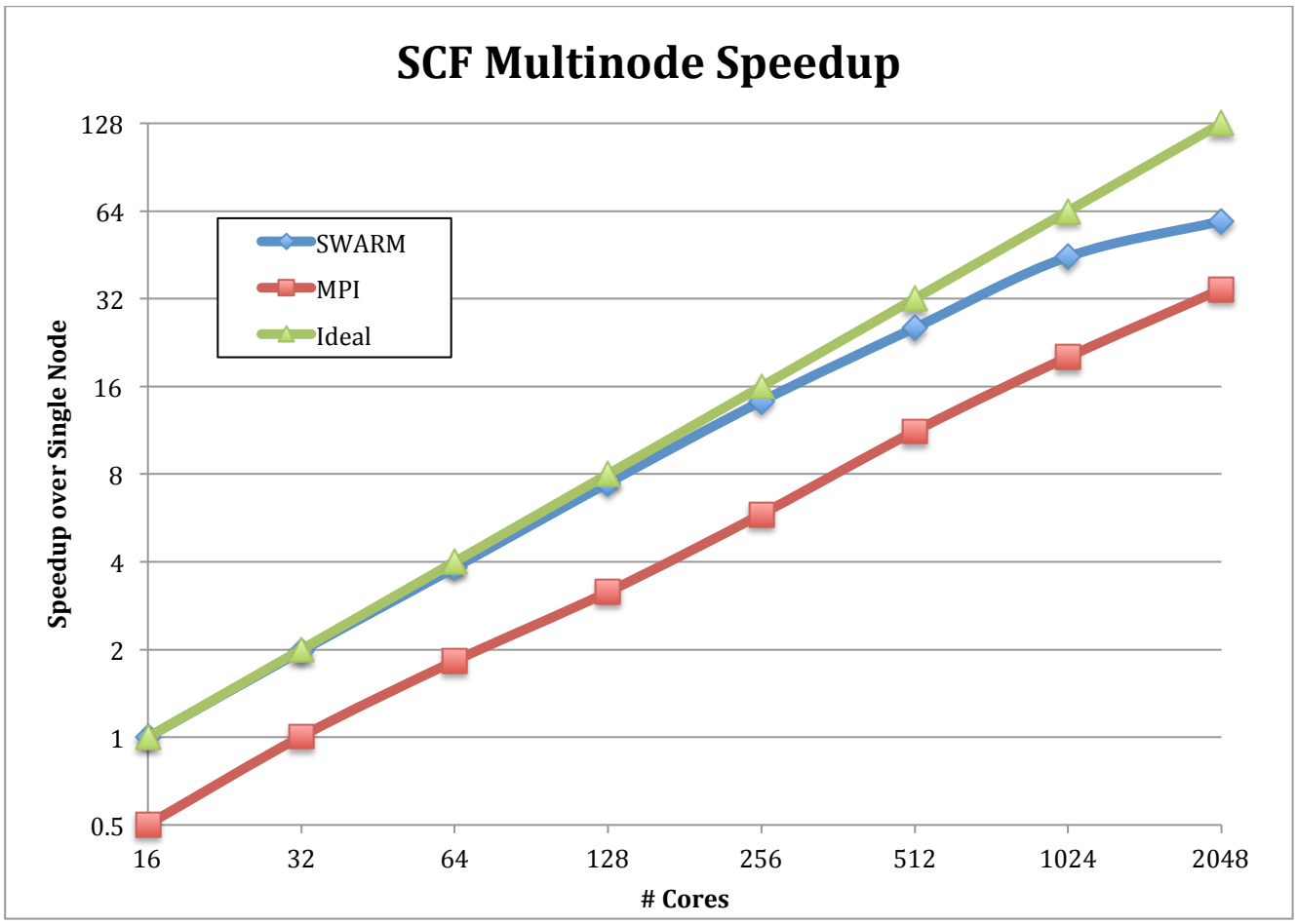
SWARM

Starting from the single-node SWARM version of SCF, network calls were inserted in order to split the **twoel** workload across nodes. The work is strided with the i parameter, such that, for a 2 node job, node rank 0 does i=0, rank 1 does i=1, rank 0 does i=2, etc. Outside of **twoel**, node 0 is responsible for the other phases of computation, such as **oneel**, **diagon, makden, damp,** and **dendif**. The workflow for an iteration is now:

- Node 0 performs single-electron interaction (**oneel)** work
- Each node does a subset of the **twoel** work
- Each node sums all of its per-thread partial fock matrices together
- A network reduction sums all of these per-node partial fock matrices together
- Node 0 adds the result into the main g_fock matrix
- Node 0 performs **diagon**
- Node 0 performs **makden, damp,** and **dendif**
- Node 0 broadcasts the global data to all nodes, to prepare for the next iteration
- Repeat

Results

The following chart shows the speedup of **twoel**, for 250 atoms arranged in two planes, over the same input running on a single node using SWARM. The single-node MPI version was twice as slow as the SWARM code, and both scaled fairly linearly from there. Each node is a 16 core machine, running 32 threads.



In the SWARM implementation, the parallelization of twoel starts to drop off at around 512-1024 cores, as the overhead of network synchronization increases. As the number of nodes increase, the real work to do per node decreases, so the percentage of total time spent doing networking increases. For larger node counts, network implementation details and broadcast/reduction parameter values start to have a large effect on overall performance.

Summary

We have performed serial optimizations to the twoel kernel of the SCF module. We have parallelized those and shown better speedups over OpenMP and MPI in most cases. The diagon kernel is now the most significant part of the module execution and we will look for ways to improve its performance in the next quarter. The original code, all optimizations, and all parallelizations are available in open source at https://www.xstackwiki.com/index.php/DynAX#Deliverables.

Reservoir: Automating hand-optimizations of SCF

We have worked on designing how R-Stream could be used to perform the most impactful optimizations performed by hand by ETI. We have looked at taking advantage of symmetry in the g() function, and the pre-computation of sub-expressions of g().

g() is evaluated up to $\mathrm{maxiter} \times \mathrm{n}^4$ times in the Hartree-Fock two-electron computation (**twoel()**), which makes it a computational bottleneck in the sequential code. Here n is the the number of atoms and $\mathrm{maxiter}$ is the maximum number of iterations of the outer time loop. The point of employing R-Stream to perform these optimizations is that R-Stream also performs automatic parallelization. The goal is then to have both parallelization and optimized sequential performance. To simplify the presentation here, we elude the time loop (maxiter).

Taking advantage of symmetry in the g() function

ETI identified symmetries in the g function, which is evaluated for every (i,j,k,l) in $[0, \mathrm{n})^4$(within a 4-deep loop nest). These symmetries are as follows:

- g(i,j,k,l) = g(k,l,i,j)
- g(i,j,k,l) = g(j,i,k,l)
- g(i,j,k,l) = g(i,j,l,k)

Each of these symmetries can be used to divide the number of evaluations of distinct values of g in half. They can be combined to divide the number of distinct evaluations by eight. Here we explain an automatic process through which this can be performed in R-Stream. The process is based on partitioning the set of evaluations using linear inequalities. For sets that represent about 7 out of 8 evaluations (i.e., loop iterations), g(i,j,k,l) is replaced with one of its equivalent values computed in the remaining 1-out-of-8 iterations.

The values can be cached to take advantage of this reuse of computed value.
Let us see how the partitioning process works for one symmetry, and then how partitions can be combined.

Case of a single symmetry

In this section we choose to consider the first symmetry because it is less simple than the two first ones. We can express the symmetry as follows:

$$\mathrm{g(i,j,k,l)} \;=\; \mathrm{g(i',j',k',l)} \text{ with } \{\mathrm{i = k', j = l', k = i', l = j'}\}, \text{ (1)}$$

where$\{0 \le \mathrm{i} \le \mathrm{n}, 0 \le \mathrm{j} \le \mathrm{n}, 0 \le \mathrm{k} \le \mathrm{n}, 0 \le \mathrm{l} \le \mathrm{n}\}$.

For clarity, from now on we will omit these boundary constraints on (i,j,k,l) (and the constraints they imply on i',j',k',l') when expressing domains of values. For instance, $\{\mathrm{i} \le \mathrm{k}\}$ should be interpreted as $\{0 \le \mathrm{i} \le \mathrm{n}, 0 \le \mathrm{j} \le \mathrm{n}, 0 \le \mathrm{k} \le \mathrm{n}, 0 \le \mathrm{l} \le \mathrm{n}, \mathrm{i} \le \mathrm{k}\}$.

Our goal is to use the symmetry to partition the (i,j,k,l) space into disjoint sets for which either g(i,j,k,l) or g(k,l,i,j) will be evaluated. Our method starts out by forming non-disjoint but symmetric sets, and then turns the symmetric sets into disjoint sets.

g(i,j,k,l) = g(k,l,i,j) is always true for (i=k, j=l), but when i<k or j<l, it is only true because of the symmetry. Hence we are using the hyperplanes {i=k} and {j=l} as partition boundaries. We are going to look at sets that include {i=k} and {j=l} first, and then make the sets disjoint.

The hyperplanes define four sets. One of the sets (let us call it A) is defined by:
$A = \{i \leq k, j \leq l\}$
Its image through symmetry (1) is
$A' = \{k' \leq i', l' \leq j'\}$
Within this image, we can use the symmetry and replace any calls to g(i,j,k,l) with g(k,l,i,j).
The other sets are:
$B = \{i \leq k, l \leq j\}$, whose image is $B' = \{k' \leq i', j' \leq l'\}$.
$C = \{k \leq i, j \leq l\} = B'$, and
$D = \{k \leq i, l \leq j\} = A'$

The union of A, A', B and B' (two sets and their image) define the entire set of values of (i,j,k,l). Using these two hyperplanes, we would partition the values of (i,j,k,l) in four sets and eliminate half of the evaluations.

However, by using the hyperplane {i=k}, we can partition (i,j,k,l) into two non-disjoint sets as follows: $A = \{k \leq i\}, A' = \{i \leq k\}$, A' being the image of A. This partitioning is simpler and results in the same two-fold savings.

However the sets are non-disjoint, so some values of (i, j,k,l) are in two different subsets. As a result, the current partition is ambiguous as to what to evaluate (g(i,j,k,l) or g(k,l,i,j)) for these values.
We can remove the ambiguity by computing their "disjoint union". The algorithm available in Jolylib (Reservoir's polyhedral library) does so by computing the (Galois) subset lattice of the input domains. The meet operation here is the intersection. Disjointness is then obtained by assigning each element (each intersection) to a unique parent in the lattice, and subtracting it from all its ancestors.
In our example, we get:
$A_1 = \{k \leq i\}: g(i,j,k,l) \rightarrow g(i,j,k,l)$
$A'_1 = \{i < k\}: g(i,j,k,l) \rightarrow g(l,k,i,j)$,
Assuming the following original code,

```
for (i=0; i<= n; i++) {
  for (j=0; j<=n; j++) {
    for (k=0; k<=n; k++) {
      for (l=0; l<=n; l++) {
        f(i,j,k,l) = g(i,j,k,l);
      }
    }
  }
}
```

The partitioning defines the following transformed code:

```
for (i=0; i<=n; i++) {
  for (j=0; j<=n; j++) {
    for (k=0; k<=i; k++) {
      for (l=0; l<=n; l++) {
        f(i,j,k,l) = g(i,j,k,l);
      }
    }
    for (k=i+1; k<=n; k++) {
      for (l=0; l<=n; l++) {
```

```
        f(i,j,k,l) = g(k,l, i,j);
      }
    }
  }
}
```

There are degrees of liberty in which we can assign g(i,j,k,l) or g(k,l,i,j) to the partition sets.

Combining symmetries

Let us now consider the three symmetries together. A naive method would be to compute disjoint partitions and combine the partitions by simple superposition. The drawback of doing so is that the algorithm that computes the disjoint partitions does not offer any form of guarantee that the boundaries of the domains formed by the different symmetries would match or mesh particularly well. Instead, we combine the symmetries by computing the symmetric, non-disjoint sets for each symmetry, combine the sets, and finally compute a disjoint partition from these sets.
For example, combining the two first symmetries would proceed as follows.
We have the following domains for the first symmetry:

For the second symmetry, we have:
$\mathrm{C} = \{\mathrm{j} \leq \mathrm{i}\}: \mathrm{g}(\mathrm{i},\mathrm{j},\mathrm{k},\mathrm{l}) \rightarrow \mathrm{g}(\mathrm{i},\mathrm{j},\mathrm{k},\mathrm{l})$
$\mathrm{C}' = \{\mathrm{i} \leq \mathrm{j}\}: \mathrm{g}(\mathrm{i},\mathrm{j},\mathrm{k},\mathrm{l}) \rightarrow \mathrm{g}(\mathrm{j},\mathrm{i},\mathrm{k},\mathrm{l})$

Combining these two symmetries gives us four domains:
$\mathrm{AC} = \{\mathrm{k} \leq \mathrm{i}, \mathrm{j} \leq \mathrm{i}\}: \mathrm{g}(\mathrm{i},\mathrm{j},\mathrm{k},\mathrm{l}) \rightarrow \mathrm{g}(\mathrm{i},\mathrm{j},\mathrm{k},\mathrm{l})$
$\mathrm{A}'\mathrm{C} = \{\mathrm{i} \leq \mathrm{k}, \mathrm{j} \leq \mathrm{i}\}: \mathrm{g}(\mathrm{i},\mathrm{j},\mathrm{k},\mathrm{l}) \rightarrow \mathrm{g}(\mathrm{k},\mathrm{l},\mathrm{i},\mathrm{j})$
$\mathrm{AC}' = \{\mathrm{k} \leq \mathrm{i}, \mathrm{i} \leq \mathrm{j}\}: \mathrm{g}(\mathrm{i},\mathrm{j},\mathrm{k},\mathrm{l}) \rightarrow \mathrm{g}(\mathrm{j},\mathrm{i},\mathrm{k},\mathrm{l})$
$\mathrm{A}'\mathrm{C}' = \{\mathrm{i} \leq \mathrm{k}, \mathrm{i} \leq \mathrm{j}\}: \mathrm{g}(\mathrm{i},\mathrm{j},\mathrm{k},\mathrm{l}) \rightarrow \mathrm{g}(\mathrm{l},\mathrm{k},\mathrm{i},\mathrm{j})$

We can now turn this non-disjoint partition into a disjoint partition:
$\mathrm{AC}_1 = \{\mathrm{k} \leq \mathrm{i}, \mathrm{j} \leq \mathrm{i}\}: \mathrm{g}(\mathrm{i},\mathrm{j},\mathrm{k},\mathrm{l}) \rightarrow \mathrm{g}(\mathrm{i},\mathrm{j},\mathrm{k},\mathrm{l})$
$\mathrm{A}'\mathrm{C}_1 = \{\mathrm{i} < \mathrm{k}, \mathrm{j} \leq \mathrm{i}\}: \mathrm{g}(\mathrm{i},\mathrm{j},\mathrm{k},\mathrm{l}) \rightarrow \mathrm{g}(\mathrm{k},\mathrm{l},\mathrm{i},\mathrm{j})$
$\mathrm{AC}'_1 = \{\mathrm{k} \leq \mathrm{i}, \mathrm{i} < \mathrm{j}\}: \mathrm{g}(\mathrm{i},\mathrm{j},\mathrm{k},\mathrm{l}) \rightarrow \mathrm{g}(\mathrm{j},\mathrm{i},\mathrm{k},\mathrm{l})$
$\mathrm{A}'\mathrm{C}'_1 = \{\mathrm{i} < \mathrm{k}, \mathrm{i} < \mathrm{j}\}: \mathrm{g}(\mathrm{i},\mathrm{j},\mathrm{k},\mathrm{l}) \rightarrow \mathrm{g}(\mathrm{l},\mathrm{k},\mathrm{i},\mathrm{j})$

The resulting code would be:

```
for (i=0; i<=n; i++) {
  for (j=0; j<=i; j++) {
    for (k=0; k<=i; k++) {
      for (l=0; l<=n; l++) {
        f(i,j,k,l) = g(i,j,k,l);
      }
    }
    for (k=i+1; k<=n; k++) {
      for (l=0; l<=n; l++) {
        f(i,j,k,l) = g(k,l,i,j);
      }
    }
  }
```

```
  for (j=i+1; j<=n; j++) {
    for (k=0; k<=i; k++) {
      for (l=0; l<=n; l++) {
        f(i,j,k,l) = g(j,i,k,l);
      }
    }
    for (k=i+1; k<=n; k++) {
      for (l=0; l<=n; l++) {
        f(i,j,k,l) = g(l,k,i,j);
      }
    }
  }
}
```

Notice that we chose to combine the symmetries in the (first, second) order. By choosing the (second, first) order we would have obtained
$A'C' = \{i \leq k, i \leq j\}: g(i,j,k,l) \rightarrow g(k,l,j,i)$
By combining the symmetries in a specific order, we took advantage of the symmetries but not of the fact that both symmetries commute (they can be applied legally in any order).

Also, while the second symmetry defines the partitioning unambiguously, the first symmetry has two degrees of freedom in terms of partitioning hyperplanes, and we have only used one. However, while using the additional degree of freedom does partition the space further, its smallest set occupies a quarter of the (i,j,k,l) domain. So the resulting code is more complex for no additional savings (a four-fold saving as well).

Optimization notes

Spatial locality
Were the g function to materialize as an array, applying the symmetries as we have has an impact on the locality of array accesses. In this case, the spatial locality of the sets A'C and A'C' is reduced. Complementary optimizations such as tiling may reduce the impact of these issues on the overall performance by reducing the number of large strides performed in the innermost loops (l loop in g(l,k,i,j) for instance).

But reducing the overall footprint of g also directly reduces the number of cache misses, which reduces the negative effect of non-spatially local accesses.

More symmetries in Hartree-Fock

Using the three symmetries, we can reduce the number of distinct values of g evaluated in the execution of the program by a factor of 8.

ETI took advantage of more symmetries than the ones present in pure function g. ETI also used symmetry in the read-only array g_dens, and in the write-only array g_fock.

It is worth exploiting symmetries in a write-only array since it reduces the number of computations that are then assigned to the array. But additional caution is needed. If we know where the array is used and if it stays symmetrical, then we don't need to complete the array definition by copying the

computed data to their symmetric, non-computed counterparts. This works as long as symmetry is used when accessing the array as well.

Otherwise, uncomputed array elements must be defined by copying their value from their symmetric computed array element.

Precomputing reused sub-expressions

Another important optimization of the bottleneck g() function performed by ETI comes from the fact that some of its expressions take n^2 distinct values within the $maxiter \times n^4$ loops in which g() is called, where maxiter is the maximum number of iterations of a time outer loop that encloses the four-deep loop calling g() as modeled in the previous section. Hence each of these expressions is re-computed $maxiter \times n^2$ times too many.

In their hand-optimization effort, ETI took advantage of this by pre-computing these sub-expressions and using the pre-computed values in the computation of instead of recomputing them. In this section, we sketch how most of this optimization can be automated in a compiler.

For presentation purposes and without loss of generality, we are eluding the outer time loop. The same optimizations extend to the outer time loop.

Basic idea

Here we explain the general idea of the proposed optimization in terms of compiler optimizations, in order to show how they could be automated.

The basic principle is to detect expressions that are pure functions (i.e., there is no side-effect within the expression) that depend upon a subset of the loop indices. In the case of Hartree-Fock, in the four-deep loop nest that calls g(i,j,k,l), we can single out a subexpression of g that depends on (i,j) only and write g(i,j,k,l) = g0(g1(i,j),i,j,k,l).

We will describe a way of detecting such subexpressions in the next section.
R-Stream is based on a scalar compiler that supports the SSA (Static Single Assignment) form, and the GVN-GCM optimization (Global Value Numbering - Global Code Motion). GVN-GCM uses SSA to assign a non-ambiguous value to each expression in the program. This is used to detect that two expressions are equivalent (which triggers common subexpression elimination) and to detect whether an expression is loop-invariant. When it determines that an expression is loop-invariant, GVN-GCM hoists the expression out of the loops.

In our example:

```
for (i=0; i<= n; i++) {
  for (j=0; j<=n; j++) {
    for (k=0; k<=n; k++) {
      for (l=0; l<=n; l++) {
        f(i,j,k,l) = g0(g1(i,j),i,j,k,l);
      }
    }
  }
}
```

In this favorable example, g1(i,j) is invariant within the k and l loops. Hence an existing GVN-GCM would hoist it above the k loop:

```
for (i=0; i<= n; i++) {
  for (j=0; j<=n; j++) {
    t = g1(i,j);
    for (k=0; k<=n; k++) {
      for (l=0; l<=n; l++) {
        f(i,j,k,l) = g0(t,i,j,k,l);
      }
    }
  }
}
```

However, GVN-GCM is insufficient when we find a pure subexpression that only depends on k and l:

```
for (i=0; i<= n; i++) {
  for (j=0; j<=n; j++) {
    for (k=0; k<=n; k++) {
      for (l=0; l<=n; l++) {
        f(i,j,k,l) = g0(g2(k,l),i,j,k,l);
      }
    }
  }
}
```

gives

```
for (i=0; i<= n; i++) {
  for (j=0; j<=n; j++) {
    for (k=0; k<=n; k++) {
      for (l=0; l<=n; l++) {
        t = g2(k,l);
        f(i,j,k,l) = g0(t,i,j,k,l);
      }
    }
  }
}
```

After a pass in the scalar optimizer, R-Stream lifts the representation of loop programs into a representation specialized on dense loops, in which all array accesses are explicit. This representation (called “polyhedral”) is based on linear algebra, but some more traditional loop optimization concepts are available there as well. The part of R-Stream which works on the polyhedral representation is called the (polyhedral) mapper.

The steps of the optimization that would hoist pre-computations are as follows:

1. Detect pre-computable statements
2. Apply fission to move the pre-computable statements before all their uses in the computation (this usually turns scalars into arrays)
3. Project the data domain along the reuse space defined by the pure/read-only function to obtain a single definition for each distinct pre-computed value.
4. Project the iteration domain along the same reuse space.

Detecting pre-computable statements

Pre-computable statements are statements that represent N definitions of values as a read-only (or pure) function of M values, where M's complexity is lower than N's. In the case of t=g1(k,l), the complexity of N is $n^4$and M's is n^2.

Applying fission

The mapper takes into account the notion of "scope" of a value, which is roughly the set of loops that contain the entire value flow for each element of an array or scalar. The mapper uses this information to correctively disambiguate ("expand") arrays automatically when performing loop transformations. The result of fission in our example is

```
for (i=0; i<= n; i++) {
  for (j=0; j<=n; j++) {
    for (k=0; k<=n; k++) {
      for (l=0; l<=n; l++) {
        t[i][j][k][l] = g2(k,l);
      }
    }
  }
}
for (i=0; i<= n; i++) {
  for (j=0; j<=n; j++) {
    for (k=0; k<=n; k++) {
      for (l=0; l<=n; l++) {
        f(i,j,k,l) = g0(t[i][j][k][l],i,j,k,l);
      }
    }
  }
}
```

Notice that the scalar t is now an array, which results from the mapper's automatic memory disambiguation (also called "expansion") mechanism. Note that the set of legal fissions is defined by the data dependencies in the program.

Projecting the data domain

The reuse space is defined by the kernel of the space of values of the read-only/pure function: $\left(\begin{matrix}1&0&0&0\\0&1&0&0\end{matrix}\right)$

Projecting the data domain along this function gives the following program:

```
for (i=0; i<= n; i++) {
  for (j=0; j<=n; j++) {
    for (k=0; k<=n; k++) {
      for (l=0; l<=n; l++) {
        t[k][l] = g2(k,l);
      }
    }
  }
}
for (i=0; i<= n; i++) {
  for (j=0; j<=n; j++) {
    for (k=0; k<=n; k++) {
      for (l=0; l<=n; l++) {
        f(i,j,k,l) = g0(t[k][l],i,j,k,l);
```

```
      }
    }
  }
}
```

Projecting the iteration domain

Finally, the redundant iterations of the precomputing statement can be removed by applying the same projection, giving:

This gives the (desired) following program:

```
for (k=0; k<=n; k++) {
  for (l=0; l<=n; l++) {
    t[k][l] = g2(k,l);
  }
}
for (i=0; i<= n; i++) {
  for (j=0; j<=n; j++) {
    for (k=0; k<=n; k++) {
      for (l=0; l<=n; l++) {
        f(i,j,k,l) = g0(t[k][l],i,j,k,l);
      }
    }
  }
}
```

Detecting pure sub-expressions

This optimization relies on the detection of pure/read-only sub-expressions in a function (g() in the case of Hartree-Fock) that depend upon a strict subset of the loop iterators.

Within R-Stream, this decomposition is best performed in the scalar optimizer.

The inputs to the optimized function may be induction variables (loop iterators) or pure functions of the induction variables.

Initially, each parameter to the function that is a pure function of the induction variables gets marked with the induction variables they depend upon.

In our example:

g(int i, int j, int k, int l)

- i depends upon [i]
- j depends upon [j]
- k depends upon [k]
- l depends upon [l]

Roughly, R-Stream's internal representation (IR) is an operator graph, in which nodes are operations and edges represent value flow between two operands. The predecessors to a node are the operations producing values for the node.

The algorithm walks the successors of the nodes that depend upon loop induction variables by keeping track of which induction variables the successor node will depend upon. Walked nodes are added to the subexpression formed until a stopping criterion is met.

A successor node is associated with the union of the induction variables associated with its predecessors. Since the goal is to form pure sub-expressions that depend upon a subset of the induction variables, the aggregation of nodes into pure subexpressions stops when (before):

1. the visited node would depend upon too many loop iterations, or
2. the subexpression would contain a side-effect. This happens when a STORE is done on a variable that escapes the subexpression. If the original function in which we are forming sub-expressions is pure, this test is always false.

There is an important tradeoff between the goals of forming large sub-expressions and forming sub-expressions that depend upon a small number of induction variables.

The amount of savings obtained through this optimization are in $C \times n^k$, where k is the number of induction variables that the expression does *not* depend upon, and C is the number of flops in the subexpression. Unfortunately, the number of loop iterations is not usually available to the scalar optimizer. Hence we have to rely on heuristics. A simple heuristic is based on the comparison between the cost of accessing a pre-computed value from memory versus the cost of recomputing it. This gives us a lower bound on the size of the subexpressions to form, since their estimated cost (in cycles) should be higher than, say, a L2 cache miss. Once this threshold is met, the heuristic would keep growing the subexpression until the number of iteration variables it depends upon grows, so that the number of reuses of the pre-computed subexpression is not reduced.

In our example, at one step of the subexpression formation, the maximal subexpressions for i,j,k, and l would be:
[i]: x[i], y[i], z[i], expnt[i]
[j]: x[j], y[j], z[j], expnt[j]
[k]: x[k], y[k], z[k], expnt[k]
[l]: x[l], y[l], z[l], expnt[l]
None of these expressions are big enough to justify precomputation, hence the formation of subexpressions aggregates subexpressions further, forming maximal expressions that depend upon 2 variables:
[i,j]: (expnt[i] * expnt[j]) / (expnt[i]+expnt[j]) * ((x[i]-x[j])(x[i]-x[j])+(y[i]-y[j])*(y[i]-y[j])+(z[i]-z[j])*(z[i]-z[j]))
[k,l]: (expnt[k] * expnt[l]) / (expnt[k]+expnt[l]) * ((x[k]-x[l])(x[k]-x[l])+(y[k]-y[l])*(y[k]-y[l])+(z[k]-z[l])*(z[k]-z[l]))
The number of flops in these expressions would be bigger than the threshold, which would define the expressions as eligible for pre-computation.

Comparison with the hand-optimization

In the previous sections, we have shown how a sub-expression of g that depends on (i,j) and on (k,l) may be optimized by computing their value once only. To summarize, we decomposed g(i,jk,l) into g0(g1(i,j),g2(k,l),i,j,k,l) and we rewrote the original four-deep loop as:

```
for (k=0; k<=n; k++) {
  for (l=0; l<=n; l++) {
    t2[k][l] = g2(k,l);
  }
}
for (i=0; i<= n; i++) {
  for (j=0; j<=n; j++) {
```

```
    t1 = g1(i,j);
    for (k=0; k<=n; k++) {
      for (l=0; l<=n; l++) {
        f(i,j,k,l) = g0(t1, t2[k][l], i,j,k,l);
      }
    }
  }
}
```

ETI detected further that g1(x,y)=g2(x,y) for any$(x,y) \in [0,n)^2$, which allowed them to not pre-compute both values of g1 and g2. Instead, they are computing one single pre-computation array:

```
for (k=0; k<=n; k++) {
  for (l=0; l<=n; l++) {
    t[k][l] = g1(k,l);
  }
}
for (i=0; i<= n; i++) {
  for (j=0; j<=n; j++) {
    for (k=0; k<=n; k++) {
      for (l=0; l<=n; l++) {
        f(i,j,k,l) = g0(t[i][j], t[k][l], i,j,k,l);
      }
    }
  }
}
```

This saved anothern^2computations per run of the four-deep loop (in reality, in both cases the precomputation happens out of the outer time loop), as compared to what our optimization would have obtained. The savings in terms of space is of only one element (t1=g1(i,j) isn't fissioned, and hence not expanded into an array).

General remarks

The four-dimensional loop that we discussed in this section is actually embedded in a time iteration loop, which has a maximum number of iterations of (say) maxiter. The savings we can obtain (and that ETI has obtained) through this technique are then not in n^2but $(\text{maxiter} \times n^2)$.

In general, a subexpression may be read-only only within a given scope (its inputs would be modified outside of that scope). The automatic optimization presented here is applicable to the restricted scope where the subexpression is read-only.

g() caching optimization

We are working on designing the automation of the optimization applied by ETI that consists of caching the values of g(). Our current thinking is that it would combine a pre-computation of g() into an array t that would be declared sparse. We would leverage the generation of sparse array codes within R-Stream that we are working on in the frame of the DoD-sponsored ENSIGN project. We will present our results in future reports.