Yearly Project Progress Report

DynAX: Innovations in Programming Models, Compilers and Runtime Systems for Dynamic Adaptive Event-Driven Execution Models

Award Number: DE-SC0008716
Dates of Performance: 9/1/2012- 8/31/2013

Principal Investigator
Rishi Khan
ET International Inc
100 White Clay Center Dr, Newark DE 19711

Co-PIs:
Benoit Meister, Reservoir Labs, Inc
David Padua, University of Illinois Urbana Champaign
John Feo, Pacific Northwest National Laboratories

Introduction

This report outlines the work that was done in the first three quarters by the DynAX Team (ET International, Reservoir Labs, UIUC, and PNNL) and work expected to be completed in the fourth quarter.

Accomplishments

The accomplishments for the year are broken down into four sections corresponding to accomplishments of each of the four teams: ETI, Reservoir, UIUC, and PNNL.

PNNL Accomplishments

Applications and performance studies

We identified two NWChem modules to server as representative benchmark for Exascale chemistry applications: 1) Self-Consistent Field Method, and 2) Coupled Cluster method. We have completed and released the first and are preparing the second. Our release includes source code, makefile, input file, output file, and optimization suggestions. We prepared a powerpoint presentation and Word document describing the mathematics, control flow, and data structures of SCF.

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, assuming that each particle of the system is subjected to the mean field created by all other particles. The solution to the Schrodinger Equation reduces to the following self-consistent eigenvalue problem

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

$$F_{\mu\upsilon}\, C_{k\upsilon} = \varepsilon\, S_{\mu\upsilon} C_{k\upsilon} \qquad [2]$$

$$D_{\mu\upsilon} = \sum_{k} C_{\mu k} C_{\upsilon k} \qquad [3]$$

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 1 are the two-electron Coulomb forces and the two-electron Exchange forces, respectively.

Figure 1 depicts the control flow of the code we released. The upper two leftmost modules initialize the ***D*** and ***C*** matrices allowing the first iteration to compute Equations 1 and 2. The *Construct Fock Matrix, Compute Orbitals,* and *Compute Density Matrix* modules compute Equations 1, 2, and 3, 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 simple rectangular, nested for loops of the form. For the most part, the loops are embarrassingly parallel, but the inclusion of reduction operations in some loops requires concurrent atomic updates. The most computationally intensive routine is **twoel** that computes the two electron forces. 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. As explained in the optimization file released with module, symmetries in ***g*** can be exploited to minimize storage and computation requirements.

The second module, the Coupled Cluster method, is interesting in that the code is generated automatically from tensor equations supplied by the user. Each equation is expressed as function called from a driver that maneuvers a DAG of the data dependencies. DAG presents a variety of scheduling alternatives and when coupled with the unbalanced data parallel character of the tensor operations and data locality issues provides a rich environment for optimization and evaluation of emerging Exascale execution models.

RPDTA for PEDAL

For the most part our work on a Power Efficient Data Abstraction Layer (PEDAL) for Rescinded Primitive Data Type Access (RPDTA) is being funded by the Traleika Glacier XStack Project (see Project Progress Report FWP PNNL-62464). Under this project we are considering only modifications required by Brandywine, and collaborating with UDEL to define a set of introspection options deemed useful for the FSIM simulator maintained by ETI.

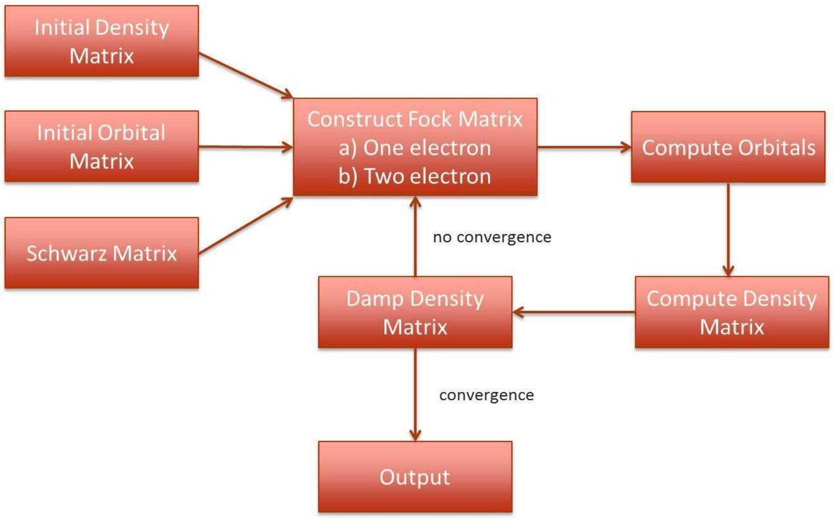

Figure 1 – SCF control flow

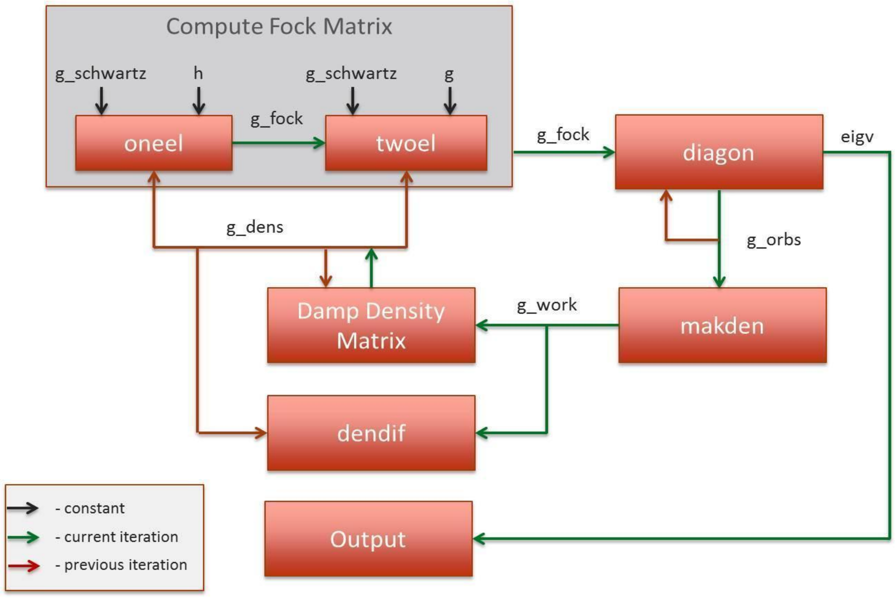

Figure 2 – SCF modules, inputs, and outputs

ETI Accomplishments

Cholesky Decomposition

We used this application to study issues of tiling, scheduling and memory management at scale. Starting with a basic implementation on SWARM, we made significant scalability improvements, and studied ways to balance the workload at runtime.

Data Distribution & Work Imbalance

The input and output matrices are symmetric, and only half of the data is stored for space efficiency reasons. We found that simple round-robin assignment of tile data to nodes leads to significant workload imbalance, and found ways to produce a more even workload balance.

Scheduling: Task Prioritization

Cholesky has a complex inter-tile dependency DAG, and this DAG spans nodes. We found that it is not sufficient for a compute node to simply execute tasks in FIFO order, and it is not sufficient to prioritize tasks based on task type. We found that a priority scheme which encourages the compute nodes to traverse the matrix in a certain order (top to bottom, or left to right) causes parallelism to be exposed much more consistently, and greatly reduces the potential for compute node starvation issues.

Memory Usage for Intermediate Data

As the node count increases, the ratio of data considered “remote” vs. “local” increases greatly. This “remote” data is often an input dependency for local computation, but as the node count increases, it quickly becomes impossible to store copies of all of the remote data locally at the same time. We implemented a form of pre-fetch, which gets the right data at the right time to keep the local computation busy without running out of memory.

We first implemented pre-fetch by fetching one row of remote data at a time, and attempting to stay a certain number of tile rows ahead of the computation. Because the matrix is symmetric, the tile rows vary in length, and this approach required careful tuning. A later and better approach was to keep track of how much memory the compute node has, and fetch new data whenever it will fit.

Results

Results and further details can be found in the Q1 report here: https://www.xstackwiki.com/index.php/File:BrandywineXStackReportQ1.docx and https://www.xstackwiki.com/index.php/File:Q1-cholesky.pptx.

SCF

We optimized and parallelized the SCF (Self-Consistent Field) code provided by PNNL, and explored some ways to improve it even further. These improvements are discussed below, organized by the math kernels involved in computing SCF.

Twoel

This math kernel computes the two-electron interactions, and is implemented as an n^4 series of nested loops. It is by far the most computationally intensive element of the program.

Serial Optimizations

We improved the performance of **twoel** on a single thread in many ways, including:

- Additional sparsity tests to skip loop iterations
- Used many symmetries in the computationally expensive **g**() function, to reduce the number of times it must be called
- Used lookup tables to increase the performance of **g**()
- Adapted several linear algebra operations (such as matrix multiply, and finding eigenvalues) to use BLAS/LAPACK
- Using symmetry of the "dens" and "fock" arrays, to reduce the amount of memory access
- Caching the output of **g**() for use in subsequent iterations

Single-Node Parallelization

We implemented two multi-threaded versions of **twoel**, one using OpenMP and one using SWARM. **I**n both implementations, each thread enumerates a subset of the n^4 iteration space, and generates its own copy of the "fock" matrix containing the results of those subsets. Those copies of "fock" are summed together at the end to produce the final result.

Twoel in SWARM saw linear speedup until the processor core count was reached, and then a slower rate of performance increase until the hyperthread count was reached.

Multi-Node Parallelization

We implemented two multi-node versions of **twoel**, one using MPI and one using SWARM. We tested these implementations at up to 128 nodes; on 128 nodes, the

SWARM version of **twoel** executed faster than the single-node implementation of **diagon**.

At higher node counts, uneven workload distribution starts to become a major factor - some nodes may spend 25% of the total **twoel** execution time waiting for other nodes to finish. We studied ways to reduce the impact of this imbalance.

Diagon

At large node counts, and with the above optimizations in place, single-node **diagon** actually takes longer than **twoel**. This function is challenging to distribute, as it is made up of multiple BLAS/LAPACK operations with different data dependency patterns. We started to study some ways to distribute it effectively across multiple nodes.

Results

Results and further details can be found in the Q2 and Q3 reports: https://www.xstackwiki.com/index.php/File:BrandywineXStackReportQ2.pdf and https://www.xstackwiki.com/index.php/File:BrandywineXStackReportQ3.pdf.

Reservoir Accomplishments

Compiler optimization design for SCF

We considered some of the most impactful optimizations performed by ETI on the SCF code and defined how they would be implemented in a compiler that supports both polyhedral loop optimization and traditional SSA compiler analysis like R-Stream.

Automating the use of symmetry in input data

We demonstrated how known symmetries in the input data can be used by a polyhedral loop optimizer to reduce the number of evaluations of the input data by up to 2^n, where n is the number of independent symmetries. The method computes a partition of the loop iterations, where every set is associated with a different expression of the input data access function.

Automating the precomputation of pure function values

We showed how to automate the pre-computation of heavily-reused sub-expressions in loop codes by combining polyhedral scope analysis, used to create the pre-computation arrays, with traditional subexpression analysis to form side-effects-free subexpressions (whose computation can be hoisted). We defined an algorithm that detects large-enough subexpression and trades off subexpression size for number of loop indices the subexpression depends upon. Depending upon fewer loop indices entails more occurrences of execution time saving (through pre-computation), while forming a larger sub-expression entails more execution time saving at each occurrence.

Automatic parallelization to SWARM (single-node, untuned)

We first implemented generic support for parallelization to codelet-based codes in R-Stream's polyhedral mapper. This consisted in defining the expected behavior of the polyhedral mapper when the target execution model is based on codelets.

To point out the main ones:
- The need for creating explicitly parallel loops disappears, and
- Codelets and inter-codelet dependencies need to be explicitly declared.

We tested these codelet-generic aspects using a back-end to OCR (funded in a separate project).
Then we used this generic codelet support to generate SWARM code. In our SWARM backend, dependence polyhedrons become "put"s in codelets producing the dependence, and symmetrically become "get"s in codelets consuming the dependence.
Asynchronous gets are supported as well through the R-Stream SWARM runtime layer, which keeps track of dependences that haven't been satisfied and yields control if a codelet needs to re-schedule itself.
We also support the generation of SWARM macros for defining codelet descriptors.

Dependence computation simplification

We identified and tackled the main bottleneck in generating codelet-based code using polyhedral techniques: the computation of inter-codelet dependence polyhedrons. We developed techniques for creating simpler dependence polyhedrons by:
- using approximations based on loop type information (doall, permutable) to bypass heavy-weight dependence computation, and
- leveraging groups of uniformly generated references to reduce the number of convex dependence polyhedrons to compute.

We also simplified the compile-time computation of dependence polyhedrons by postponing some of their evaluation to run-time.

UIUC Accomplishments

Various enhancements to the Parallel Intermediate Language (PIL) were implemented during the first three quarters and an evaluation of Cholesky decomposition in PIL was carried out. Enhancements affected the PIL libraries and the language semantics for data management. New capabilities include: tiled array data representation, Structured PIL (SPIL) extension, and SPMD PIL.

To evaluate the effect of overhead on PIL programs, a study using the Cholesky decomposition was implemented in PIL and the execution time of the PIL version was compared with a hand-coded SWARM implementation. The PIL code targeting both SWARM and OpenMP show scalability with increased number of threads and little execution overhead.

Library functions can now be written as collections of PIL nodes. At the beginning of the project, PIL programs had a flat organization and contained a collection of nodes linked by dependence arcs to form a parallel program written in terms of codelets. With the new extension, one can now also implement a parallel kernel as a subgraph of PIL nodes and include it as a library function. In this way, it is now possible to enter a kernel as a subgraph at any point in the execution of a PIL program, suspend the execution of the program, perform the desired operation by executing the codelets in the order enforced by the dependences, and return execution control to the program that made the request at the end of the kernel. Hierarchically Tiled Arrays (HTAs) were implemented using this new PIL library feature. Our new PIL library feature facilitates reusability of common parallel operations and algorithms.

Supports for a data structure for creating and manipulating tiled arrays, as well as fundamental parallel operations on tiled arrays were also designed. Our design allows dense arrays to have a row-major or tile-major layout for different optimization requirements. Operations for expressing data associations with codelets as well as data movement operations were also designed. These extensions provide programmers with high level representation and operations and the ability to manipulate the data either automatically by compiler or manually to optimize data movement.

The design of a high level language SPIL was also completed. It provides syntactic sugar for common operations in PIL for programming convenience and code readability. The SPIL will have a source-to-source translator to generate code in PIL. With SPIL, programmers can express their parallel algorithm more easily and more intuitively than using PIL.

For distributed memory environment, a new design SPMD PIL was implemented targeting distributed SCALE. With this extension, PIL can be programmed in an 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. Barrier API is also implemented to allow explicit synchronization across PEs and enforce an ordering of execution intended by programmers. In this model, software designers have a finer-grained control to communications in distributed memory environment, which permits better performance optimization.

PIL API document is updated to version 0.4 to reflect all the new features described above. It can be found at https://www.xstackwiki.com/index.php/DynAX#Deliverables

In the final quarter, we plan to continue working on sparse data representation in PIL, since the use of sparse data sets frequently appears in scientific applications. Our goal is to deliver an easy and intuitive representation of sparse data and a high performance implementation in PIL in order to expedite application development.

Technologies Delivered

Technologies that are delivered can be found on the DynAX page of the X-Stack wiki: https://www.xstackwiki.com/index.php/DynAX#Deliverables

Technologies Delivered (Q1-Q3):

- NWChem SCF module:
    - Serial C version (PNNL-Q2)
    - Optimized C version (ETI-Q2)
    - OpenMP Version (ETI-Q2)
    - MPI Version (ETI-Q2)
    - SWARM single and multi-node Version (ETI-Q2/Q3)
- NWChem TCE module:
    - Serial C Version (PNNL-Q3)
    - OpenMP Version (PNNL-Q3)
    - CUDA Version (PNNL-Q3)
    - Fortran99 Version (PNNL-Q3)
- Cholesky Decomposition optimization for scheduling, memory management and self-awareness (ETI) (Q1)
- Stencil framework for CnC and SWARM (Reservoir-Q3)
- Single-node untuned automatic *mappable C* -> SWARM parallelization with R-Stream (Reservoir-Q3)
- Parallel Intermediate Language (PIL) -> SCALE translator
    - Single Node (UIUC-Q1/Q2)
    - Multinode (UIUC-Q3)
- PIL API Document (UIUC-Q2/Q3)

Technologies anticipated to be delivered (Q4):

- NWChem TCE Module:
    - SWARM single node version (ETI)
    - Basic single-node parallelization to SWARM using R-Stream
- R-Stream (Reservoir):
    - Tuned single-node *mappable C* -> SWARM parallelization
    - Untuned Multi-node *mappable C* -> SWARM parallelization
    - Simplified domain computation (Improvement of compiler engine tractability)
- Sparse data representation in PIL (UIUC)

Technologies in progress in Year 1 to be completed in Years 2-3:

- Preliminary design and baseline energy study for NWChem reference kernels for SWARM (PNNL)
- RPDTA for PEDAL (PNNL)
- Asynchronous Distributed Tile Operation Stack (ETI)
- R-Stream optimizations for SCF defined in Q2

- More SWARM-specific optimizations in R-Stream as we experiment with SWARM parallelization.

Presentations

- Project overview presentation: PI Kickoff meeting in September 2012.
- Project overview presentation: DOE ASCR meeting in October 2012.
- Presentation on Cholesky Decomposition enhancements in December 2012.
- Presentation to TACC on SCF optimizations in February 2013.
- Progress-to-date presentation: 6-month PI meeting in March 2013.
- X-StackProject overview and results: EXaCT all-hands meeting in May 2013.

Publications

None at this time.
We are working on publications for the Cholesky and SCF work.

Websites

The DynAX group created and maintains the X-Stack wide website for meetings, project overviews, and deliverables at: http://xstackwiki.com.

Unexpended Funds

We do not anticipate any unexpended funds for the period 9/1/2012-8/31/2013.