POSTER: Utilizing Dataflow-based Execution for Coupled Cluster Methods

Heike McCraw*, Anthony Danalis*, Thomas Herault*, George Bosilca*, Jack Dongarra*,
Karol Kowalski† and Theresa L. Windus‡
*Innovative Computing Laboratory (ICL), University of Tennessee, Knoxville
†Pacific Northwest National Laboratory, Richland, WA
‡Iowa State University, Ames, IA

I. Introduction

Computational chemistry comprises one of the driving forces of High Performance Computing. In particular, many-body methods, such as Coupled Cluster methods (CC) [1] of the quantum chemistry package NWChem [2], are of particular interest for the applied chemistry community.

With the increase in scale, complexity, and heterogeneity of modern platforms, traditional programming models fail to deliver the expected performance scalability. On our way to Exascale, we believe that dataflow-based programming models – in contrast to the control flow model (e.g., as implemented in languages such as C) – may be the only viable way for achieving and maintaining computation at scale.

In this paper, we discuss a dataflow-based programming model and its applicability to NWChem's CC methods. Our dataflow version of the CC kernels breaks down the algorithm into finer grained tasks with explicitly defined data dependencies. As a result, the serialization imposed by the traditional, linear algorithms can be transformed into parallelism, allowing the overall computation to scale to much larger computational resources. We build this experiment using the Parallel Runtime Scheduling and Execution Control (PaRSEC) framework [3] – a task-based dataflow-driven execution engine – that enables efficient task scheduling on distributed systems.

II. Coupled Cluster Theory over PaRSEC

Utilizing task scheduling systems, such as PaRSEC in order to execute NWChem's CC codes is not trivial. The CC code is neither organized in pure tasks – i.e., functions with no side-effects to any memory other than arguments passed to the function itself – nor is the control flow affine (e.g. loop execution space has holes in it; branches are statically undecidable since their outcome depends on program data, and thus it cannot be resolved at compile time). However, while the CC code is neither affine, nor statically decidable, all the program data that affects the behavior of CC is constant during a given execution of the code. Therefore, the code can be expressed as a parameterized Directed Acyclic Graph (DAG), by using lookups into the data of the program.

To create a PaRSEC-enabled version of one of the >60 CC subroutines – `icsd_t1_2_2_2()` – we decomposed it into two steps. The first step traverses the execution space and evaluates all `IF` branches, without executing the actual computation kernels (`SORT`s, matrix-multiply kernels (`GEMM`s)). This step uncovers sparsity information by performing all the lookups into the program data that is involved in the `IF` branches prior to the computation, and stores the results in new custom meta-data vectors. Since the data of NWChem that affects the control flow is immutable at run-time, this first step only needs to be performed once.

In addition to the first step, we created a parameterized representation of the DAG called Parameterized Task Graph (PTG) [4]. This PTG includes lookups into our custom meta-data vectors populated by the first step, so that the execution of the modified subroutine over PaRSEC perfectly matches the original execution of `icsd_t1_2_2_2()`. Fig. 1 shows one chain (of a total of 12) from the DAG generated by executing the PaRSEC version of the subroutine, using *uracil-dimer* (in 6-31G basis set composed of 160 basis set functions) as the input molecule. It is clear that the execution forms a chain, where each task (a `GEMM` in particular) has to wait for the completion of the previous one (as well as the task that reads the necessary input data). In terms of parallelism and load balancing, this precisely matches the execution of the original NWChem code, where a series of `GEMM` operations, executed sequentially in a loop, constitutes a single task. In the case of *uracil-dimer* there are 12 such independent chains, each performing 24 `GEMM`s.

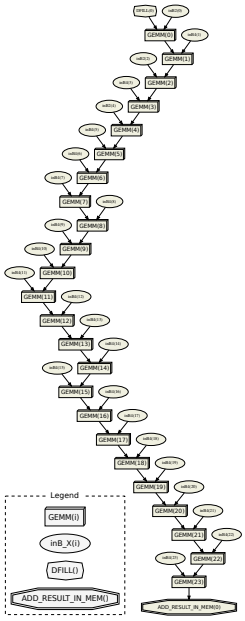

Fig. 1. Chain of 24 GEMMs

However, the most significant outcomes of porting CC over PaRSEC is (1) the ability of expressing tasks and their data dependencies at a finer granularity, and (2) the decoupling of computation and communication that enables us to experiment with more advanced communication patterns than serial chains. A `GEMM` kernel performs the operation $C = \alpha * A * B + \beta * C$ where A, B, C are matrices and α, β are scalar constants. Since matrix addition is an associative and commutative operation, the order in which the `GEMM`s are performed does not bare great significance†, as long as the results are atomically added. This enables us

†Changing the ordering of GEMM operations leads to results that are not bitwise equal to the original, but this level of accuracy is rarely required, and is lost anyway when transitioning to different compilers and/or math libraries.

978-1-4799-5548-0/14/$31.00 ©2014 IEEE

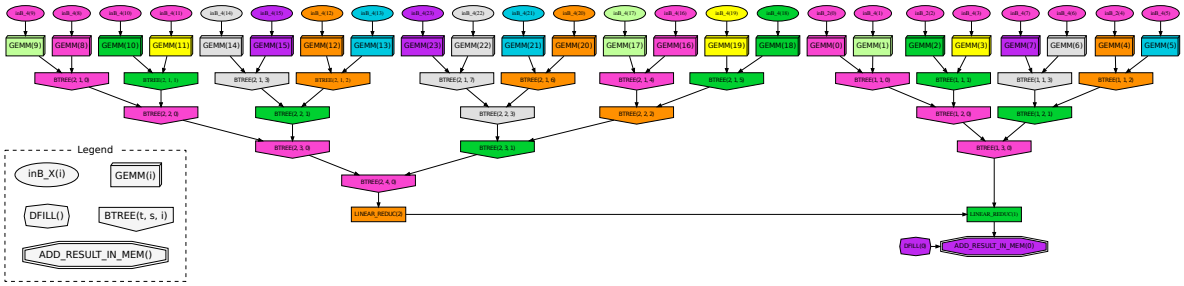

Fig. 2. Parallel GEMMs and binary reduction of results (each color corresponds to 1 out of 8 nodes that participated in this run)

to perform all GEMMs in parallel and accumulate the results using a binary reduction tree in PaRSEC. Fig. 2 shows the DAG of one of the 12 "chains" of GEMMs (*uracil-dimer* input), generated by PaRSEC with binary reduction.

Clearly, in this implementation there are significantly fewer sequential steps than in the original chain (Fig. 1). Each color in Fig. 2 corresponds to one of the eight nodes that participated in this run. While the original CC implementation treats the entire chain of GEMMs as one "task" and therefore assigns it to one node, our new implementation of `t1_2_2_2()` over PaRSEC distributes the work onto different hardware resources leading to better load balancing and the ability to utilize additional resources. That is, the PaRSEC version is by design able to achieve better strong scaling (constant problem size, increasing hardware resources) than the original code.

III. Experimental Evaluation

The performance data of `icsd_t1_2_2_2()` for the original NWChem and our dataflow-based implementation with PaRSEC is compared in Fig. 3. As input we used the beta carotene molecule in 6-31G basis set composed of 472 basis set functions. For the original version of `icsd_t1_2_2_2()`, this results in 48 chains, each computing 48 sequential GEMM's. The scalability tests were performed on the Titan Cray-XK7 computer system at Oak Ridge National Laboratory. Each node has 32 GB of RAM and one 16-core AMD Opteron (Interlagos) processor running at 2.2 GHz. We performed performance tests utilizing 1, 2, 4, 8, and 16 cores per node.

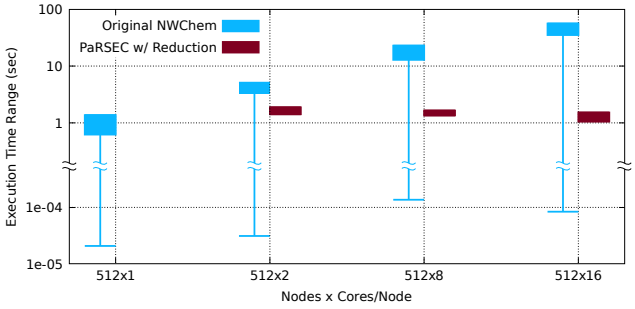

Fig. 3. Performance of original and dataflow version of NWChem's CC

Each (light) blue box represents the execution time range observed for the 48 slowest processes of the original code (i.e., only the 48 processes involved in the computation of the chains[‡]). The whiskers show the execution time of the other processes that are idle (since their execution time is on the order of $100\mu s$). Interestingly however, in the case of the original code, as we increase the number of processes per node we observe a dramatic performance degradation. The execution time of the 48 working processes (i.e., the boxes in the graph) goes from the order of $1s$ when using one core per node, to $5s$ with two cores, to $20s$ with eight and finally to over $50s$ when all the cores of each node are running NWChem processes. This behavior demonstrates the inability of the original code to utilize additional resources to speed up a fixed problem, i.e., lack of strong scaling.

The same graph depicts the behavior of our implementation of the subroutine over PaRSEC using the (dark) red boxes, which represent the entire range from minimum to maximum. PaRSEC uses one process per node and an increased number of threads per node when additional cores are being used. By design the PaRSEC version of the code uses individual GEMMs as the unit of parallelism (as opposed to a whole chain of GEMMs) and thus it is able to utilize more hardware resources. Also, PaRSEC internally uses an additional thread for performing the communication, so the minimum recommended core count per node is two.

In terms of absolute time, the PaRSEC version of the code is at best (for 16 cores/node) on a par with the best performance achieved by the original code (for 1 core/node). As can be seen in the graph, while the performance of the original code deteriorates with the increasing number of resources, the performance of the PaRSEC version improves as the number of cores per node increases. If we compare the performance of the original versus the modified code when all the cores per node (or half the cores per node) are used then our dataflow version outperforms the original by more than an order of magnitude.

[‡]However, more than 48 nodes are needed due to memory requirements.

Acknowledgment

This work is supported by the Air Force Office of Scientific Research under Award No. FA9550-12-1-0476; and used resources of the Oak Ridge Leadership Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] R. J. Bartlett and M. Musial, "Coupled-cluster theory in quantum chemistry," *Reviews of Modern Physics*, vol. 79, pp. 291–352, 2007.

[2] M. Valiev, E. J. Bylaska, N. Govind, K. Kowalski, T. P. Straatsma, H. J. J. Van Dam, D. Wang, J. Nieplocha, E. Aprà, T. L. Windus, and W. A. de Jong, "NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations," *Computer Physics Communications*, vol. 181, pp. 1477–1489, 2010.

[3] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, J. Dongarra, "DAGuE: A generic distributed DAG engine for High Performance Computing," *Parallel Computing*, vol. 38, pp. 37–51, 2012.

[4] M. Cosnard and M. Loi, "Automatic task graph generation techniques," in *HICSS '95: Proceedings of the 28th Hawaii International Conference on System Sciences*. Washington, DC: IEEE Computer Society, 1995.