

## **REFACTORING AND AUTOMATED PERFORMANCE TUNING OF COMPUTATIONAL CHEMISTRY APPLICATION CODES**

Shirley Moore

University of Texas at El Paso  
El Paso, TX 79968, USA

### **ABSTRACT**

Computational chemistry codes such as GAMESS and MPQC have been under development for several years and are constantly evolving to include new science and adapt to new high performance computing (HPC) systems. Our work with these codes has given rise to two needs. One is to refactor the codes so that it is easier to optimize them. After profiling has identified performance critical regions, refactoring to outline those regions into separate routines facilitates performance tuning and porting to complex heterogeneous HPC architectures. The second need is for automated performance tuning. Because of the large number of both fine-grained and coarse-grained parameters for tuning performance on complex hierarchical and hybrid architectures, the search space for an optimal set of parameters becomes very large. This paper describes initial results on using refactoring tools to restructure MPQC and GAMESS and on using automated tools to tune performance on multicore and manycore architectures.

### **1 INTRODUCTION**

Computational chemistry codes such as GAMESS (Schmidt et al. 1993) and MPQC (Kenny 2012) are among the most widely used simulation codes, with the current user base for these codes exceeding 100,000 users. Given the great interest in applications such as solar energy cell design, combustion efficiency, materials science, nanoscience, nanoelectronics and related fields, usage of these codes is certain to increase in the coming years. GAMESS and MPQC have been under development for several years and are constantly evolving to include new science and adapt to new high performance computing (HPC) systems. However, porting and tuning these codes will be enormously challenging, as high-performance computers are destined to greatly increase both in complexity and in system size. Within a few years, high-end systems will feature hundreds of thousands of nodes, each of which may feature 100 or more processor cores (Borkar 2011). Experience has shown that tuning techniques and procedures that are effective for one system are often completely inappropriate for another. As a result, tuning for an individual system is a highly expertise- and time-intensive process.

Our work with these codes has given rise to two needs. One is to refactor the codes so that it is easier to parallelize and optimize them. Refactoring to eliminate global variables helps make the codes thread-safe and allows better use of automatic tools such as auto-differentiation tools. After profiling has identified performance critical regions, refactoring to outline those regions into separate routines facilitates performance tuning and porting to manycore architectures such as Graphical Processing Units (GPUs) and the Intel Many Integrated Core (MIC) architecture.

The second need is for automated performance tuning. Because of the large number of both fine-grained and coarse-grained parameters for tuning performance on complex hierarchical and hybrid architectures, the search space for an optimal set of parameters becomes very large.

The remainder of this paper is organized as follows. Section 2 gives some background on computational chemistry codes. Section 3 describes source refactoring tools and how we are using them to trans-

form computational chemistry codes to make the codes more amenable to adapting and optimizing them for new HPC architectures. Section 4 describes an automated tuning framework and how we are using it to identify performance critical portions of the code and remove performance bottlenecks. Section 5 gives conclusions and outlines future work.

## 2 COMPUTATIONAL CHEMISTRY CODES

GAMESS (Schmidt et al. 1993) is a broad-based computational chemistry suite of programs that includes the highest levels of ab initio electronic structure theory, as well as innovative methods for improved scaling of electronic structure theory and novel methods for treating intermolecular forces. Parallelism in GAMESS is facilitated by the distributed data interface (DDI) that provides one-sided communication in a transparent fashion to the programmer, while allowing the programmer to make important optimizations to obtain the best performance possible in the model. Distributed parallel algorithms are enabled for Hartree-Fock, second order Møller–Plesset perturbation theory (MP2), simple levels of configuration interaction (CI), coupled cluster, multi-configurational self-consistent field (MCSCF) and multi-reference perturbation theory, including energy derivatives for many of these methods. GAMESS runs on essentially all platforms, ranging from PCs to BlueGene systems with tens of thousands of processors.

Quantum chemistry methods are available that provide essentially any level of accuracy desired by the user. However, there is often a large tradeoff – the more accurate the method, the longer the calculation takes to complete. This generalization has been changing in recent years with the use of localized orbitals, Cholesky decomposition, and other “linear scaling” methods. Of particular interest are linear scaling methods that can be applied to very complex bond making and breaking chemical reactions, as well as electronic excitations, and that require the use of multi-configurational reference states. A linear scaling multi-reference configuration interaction (MRCI) method developed at Princeton University (Oyeyemi 2011) is currently being integrated into GAMESS. The resulting code is called GAMESS+TigerCI. To the best of our knowledge, this will be the first linear scaling MRCI implementation that is designed to run on tens of thousands of cores.

The Massively Parallel Quantum Chemistry program (MPQC) (Kenny 2012) was designed to utilize distributed memory parallelism from its inception. It uses object-oriented programming techniques and is implemented in the C++ language. Parallelism is obtained in one of several ways, depending on the architecture and the method being employed. MPQC can use the Message Passing Interface (MPI) only, it can use thread-safe MPI to implement true one-sided remote memory operations (Nielson and Janssen 2008), or it can use MPI with Aggregated Remote Copy Interface (ARMCI) (Nieplocha 2006). Within each node, MPQC can use multi-threading for parallelism (Nielson and Janssen 2008). MPQC implements density functional theory, as well as advanced correlation techniques utilizing explicit electron correlation (Valeev and Janssen 2004). It also has an implementation of a local, linear-scaling correlation technique, LMP2 (Nielson and Janssen 2007), similar in spirit to the linear-scaling MRCI that is being integrated into GAMESS as discussed above. MPQC has a particularly flexible mechanism for dealing with integrals. A super-integral object can be used to mix and match integral evaluators from different integral packages.

Computational quantum chemistry code packages provide a variety of methods whose performance varies greatly and whose optimal configuration must take into account the properties of calculation approaches, parameters of the algorithms, aspects of the parallel hardware, and the particular molecular targets being solved. This configuration is necessary both prior to execution, to select the best possible code forms, and adaptively at runtime, as the dynamic performance changes. The following sections describe how we are using source refactoring and auto-tuning tools to generate optimal code implementations and configurations.

## 3 SOURCE REFACTORING

*Refactoring* changes a source program to improve its internal design but does not change its external behavior (Fowler 1999). Refactoring can range from minor style changes and readability improvements

to performance improvements (e.g., interchanging loops under certain conditions) to large-scale design changes (e.g., moving a procedure from one module to another).

Refactoring a program by hand is tedious and error-prone. Photran is an Eclipse-based integrated development environment and refactoring tool for Fortran (Watson and Debardeleben 2006). Photran has an *Extract Procedure* feature that can remove a sequence of statements from a procedure, place them into a new subroutine, and replace the original statements with a call to that subroutine. Any local variables used by those statements are then passed as parameters to the new procedure. Such a process is also called *outlining*.

Fortran 77 has no *global* variables, i.e. variables that are shared among several program units (subroutines). The only way to pass information between subroutines is to use the subroutine parameter list. Sometimes this is inconvenient, for example when many subroutines share a large set of parameters. In such cases one can use a COMMON block. This is a way to specify that certain variables should be shared among certain subroutines. In general, the use of common blocks should be minimized. Fortran allows different definitions of a COMMON block to give the same variable different names. This is confusing. Photran has a refactoring feature that gives the variables the same names in all definitions of a given COMMON block, thus improving readability and maintainability of the code.

A Photran-based tool has been developed to eliminate global variables from Fortran codes (Negara et al. 2010). The presence of global and static variables is a major obstacle to converting a legacy MPI application such as GAMESS to a hybrid MPI+threads execution model. The global variables in the MPI code need to be privatized to ensure thread safety. One approach is to manually remove global variables from the source code but this approach is tedious and error-prone.

The ROSE outliner addresses the problem of extracting tunable kernels from large scale applications (Liao et al. 2009). The outliner is based on the ROSE compiler infrastructure (Quinlan et al. 2012), which is a source-to-source compiler framework that enables building program transformation and analysis tools for large scale C/C++, Fortran, OpenMP and UPC applications. We have found however that ROSE is most robust for C and C++ programs and less so for Fortran. ROSE presents a common object-oriented, open source IR (intermediate representation) for multiple languages. The IR includes an abstract syntax tree (AST), symbol tables, and a control flow graph. The outliner is just one of the transformations that can be built on top of the ROSE IR. The outliner serves as a bridge between whole applications and existing empirical tuning methods and tools suitable for handling kernels. The outliner generates kernels that preserve performance characteristics of the tuning target as much as possible, for example by avoiding excessive pointer dereferencing in the outlined routine.

### 3.1 Refactoring GAMESS

Some initial work has been done by the GAMESS team on porting portions of GAMESS to multicore systems augmented with GPUs. As explained below in section 4.4, we are using automated methods to identify kernels in GAMESS that are expected to achieve substantial performance gains when ported to GPUs. Once these kernels are identified, we are using the Eclipse Photran interface to outline the kernels into separate routines and make them thread-safe so that they can then be ported to GPUs.

### 3.2 Refactoring MPQC

For the work described below in Section 4.2 on autotuning the MPQC integral computations, we outlined the `blockbuildprim_1()` kernel by hand. We are currently attempting to use the ROSE outliner to outline additional nested loop structures from `blockbuildprim_1()` into separate routines to serve as a basis for further autotuning efforts.

## 4 AUTOMATED PERFORMANCE TUNING

Computational quantum chemistry code provide a variety of methods whose performance varies greatly and whose optimal configuration must take into account the properties of calculation approaches, parameters of the algorithms, aspects of the parallel hardware, and the particular molecular targets being solved. This configuration is necessary both prior to execution, to select the best possible code forms, and adaptively at runtime, as the dynamic performance changes. Our approach to date has been to combine a fine-grained methodology for automated loop transformations and code specialization with a coarse-grained parameter tuning methodology for selecting optimal parameters at runtime. Our approach makes use of the SUPER autotuning infrastructure described below.

### 4.1 SUPER Autotuning Framework

The SUPER project (Lucas 2012) is developing a common framework to allow autotuning tools to share information and facilitate composition into the most appropriate set of tools for a particular application. The SUPER autotuning framework, originally developed as part of the Performance Engineering Research Institute (PERI) project (Bailey et al. 2009), is shown in Figure 1. The first step is to use performance analysis tools to identify regions of an application code that are performance bottlenecks. The Performance API (PAPI) (Browne et al. 2000) is a particularly useful tool for capturing metrics such as cycles, instruction counts, and cache and memory statistics. A performance-critical region is then outlined into a separate routine. A compiler-based approach applies code transformations to rewrite the routine from its original form to one that more effectively exploits architectural features such as registers, caches, SIMD compute engines, and multiple cores. An empirical search engine evaluates various code versions to find the optimal implementation for the underlying platform.

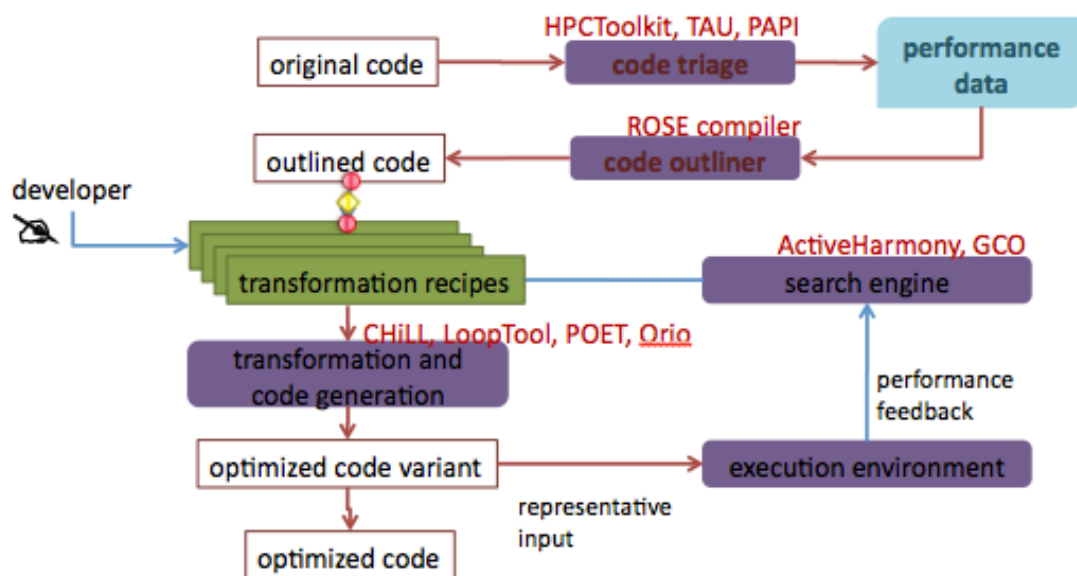


Figure 1: SUPER automatic tuning framework

### 4.2 Autotuning MPQC

An important computational component of quantum chemistry algorithms is the evaluation of the various integrals that arise in a given method. MPQC has a particularly flexible mechanism for dealing with inte-

grals. A super-integral object can be used to mix and match integral evaluators from different integral packages. Thus, MPQC was selected as an initial testbed for autotuning integral computations.

We followed the SUPER autotuning workflow described above to tune the integral computations in MPQC. Our initial profiling results, shown in Table 1, showed that the `blockbuildprim_1()` routine was taking the greatest amount of time. We instrumented the `blockbuildprim_1()` routine using PAPI to collect hardware counter data. The results are shown in Table 2. We found that the cycles per instructions (CPI) metric was quite high (CPI  $\sim 4$ ) but that L1 and L2 cache hit ratios were good. Note that 21% of all cycles were stalled, which suggested inadequate instruction level parallelism (ILP). We outlined a kernel very similar to `blockbuildprim_1()`, shown in Figure 2, in order to try both autotuning and hand tuning. `blockbuildprim_1()` actually contains several such nested loop structures. Our hypothesis was that there was insufficient instruction level parallelism (ILP). We tried removing all indirect addressing, but CPI was still around 4. We applied loop unrolling by hand to expose more parallelism. This succeeded and lowered CPI to around 1.4, even after putting back the indirect addressing. Finally, we generated specialized versions of the code for different input sizes. The final results are shown in Table 3. The variable `am34` gives the input size. Typical values of this parameter are 6 or 7, for which we achieved a significant reduction in the CPI metric.

The MPQC developers have also provided a set of user-tunable parameters for coarser-grained autotuning of the integral computations. There are a total of 10 parameters with 26244 total possible combinations of parameter values. The parameters include choices such as whether to swap the order of general contraction loops, whether or not to use redundant primitives, whether or not to use generated code, and various versions of low-level routines. We used the General Code Optimization (GCO) search engine (Seymour, You, and Dongarra 2008) to perform an exhaustive search of all the parameter combinations and achieved a 30% performance improvement over the default settings. GCO is fast and effective for performing offline searches, whereas Active Harmony also has the ability to perform online searches.

Table 1: Gprof profile of MPQC integral computation

% time	Cumulative seconds	Self seconds	#calls	name
27.97	15.10	15.10	18,157,902	<code>sc::Int2eV3::blockbuildprim_1()</code>
8.40	19.63	4.53	12,508,925	<code>sc::Int2eV3::compute_erep()</code>
6.82	23.31	3.68	12,500,000	<code>sc::FAVI.MMap&lt;&gt;::find()</code>
6.73	26.94	3.63	5,960,291	<code>do_sparse_transform2_3new()</code>
6.11	30.23	3.30	8,392,891	<code>do_sparse_transform2_1new()</code>
4.97	32.91	2.68	1,332,270	<code>sc::Int2eV3::shiftam_34()</code>
4.96	35.59	2.68	5,942,149	<code>do_sparse_transform2_2new()</code>
4.15	37.83	2.24	6,405,352	<code>sc::Int2eV3::build_not_using_gcs()</code>
3.85	39.91	2.08	2,365,269	<code>sc::Int2eV3::shiftam_12()</code>
2.71	41.37	1.47	1,250,000	<code>sc::Int2eV3::int_have_stored_integral()</code>

Table 2: PAPI data for the `blockbuildprim_1()` routine

FLOPS/Cycle	0.24 (i.e., CPI = 4.2)
L1 cache miss rate	0.45%
L2 cache miss rate	5.6%
TLB miss rate	0.017%

Branch miss prediction rate	3.7%
Cycles stalled	261 M (21% of total cycles)

```

void blockbuildprim_1(double* A2, double* B, int amin, int amax, int am34, int size34) {
    for(am12=amin; am12<=amax; am12++) {
        for (i12=2; i12<=am12; i12++) {
            for (k12=0; k12<=am12-i12; k12++) {
                double *A=&A2[am34+1];
                double d = half_ooze;
                k = 0;
                for (i34=1; i34<=am34; i34++) {
                    for (k34=0; k34<=am34-i34; k34++) {
                        A[k] += d * B[k];
                        k++;
                    }
                    d += half_ooze;
                }
                A2 += size34;
            }
        }
    }
}

```

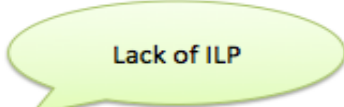


Figure 2: A stand-alone kernel extracted from the MPQC blockbuildprim\_1() routine

Table 3: Improvement in CPI metric for transformed buildblockprim\_1() kernel

Variable am34 (input case)	Old CPI	New CPI
7	4.87	1.18
6	3.60	1.19
5	2.82	1.28
4	3.57	1.49
3	3.75	1.78

### 4.3 Autotuning GAMESS

Our work with GAMESS is in its initial stages. Currently, we have collected profile data, including both timing and hardware counter data, to identify portions of GAMESS for autotuning. Hardware counter data included cycle and instruction counts and cache, memory and translation lookaside buffer (TLB) statistics. From these data, we were able to compute derived metrics for the Cycles Per Instruction (CPI) metric and a cycle breakdown to attribute cycles to various causes – for example, floating point arithmetic or waiting for memory accesses. We did the runs on the Lonestar Linux cluster at Texas Advanced Computing Center. A few of the analyses and observations that we have made concerning the execution behavior of GAMESS are summarized below.

In general GAMESS is very well optimized. For the inputs we used, the CPI metric for each important function and nested loop structure is, with one exception, 0.6 or less. There is one routine (tranot) where there are complex nested loops from four to nine levels deep. Making these loop nests simpler might provide more efficient execution. The routine belongs to the so-called AO-to-MO transformation, a 4-index array transformation commonly used to transform two-electron integrals from atomic orbital (AO) basis to molecular orbital (MO) basis, where the indices may be as large as a few thousand for large molecules. There is one routine (vclr) that has a fairly high last level cache miss percentage. This suggests that one of the arrays in the loop has a large stride or that too much data is being accessed. Coding the loop with a smaller stride will help it run more efficiently on an accelerator. Many loops are vectorized, but some loops that are not vectorized are blocked by indirect array lookups. Avoiding indirect lookups would not only assist the compiler to vectorize the code (although vectorization depends on other factors as well), but it would also make better use of processor caches and TLBs because the processor prefetch unit might be able to recognize the predictable memory accesses and hence bring them into the cache before they are accessed by the program. The floating point efficiency may benefit from using SSE2 packed arithmetic intrinsics. There are a few ways we could increase the chances of the compiler generating packed arithmetic. Specifying alignment and/or padding of data structures might help.

In the TigerCI module, profiling has identified the Cholesky decomposition step and the transformation of the Cholesky matrix from the atomic to the molecular basis as performance bottlenecks. We have observed that a loop transformation could accelerate a key part of the code by a factor of three. Our current effort is to perform these transformations via automatic tools. We also plan to develop GPU implementations of portions of the TigerCI code using the methodology described in section 4.4.

#### 4.4 GPU Implementations of GAMESS Kernels

We are using the PAPI (Browne et al. 2000), PerfExpert (Burtscher et al. 2010) and MACPO (Rane and Browne 2011) tools to identify portions of the GAMESS code that are amenable to porting to GPUs. We initially use the SUPER autotuning framework to identify the most time consuming kernels in the code and optimize these kernels for multicore architectures. Optimizing for multicore generates streaming parallelization and vectorization and enables the code to be mapped to the data parallel SIMD/SIMT execution required for GPUs. We then use PerfExpert, PAPI, and MACPO to collect hardware counter and memory access data and eliminate those kernels that are not expected to execute efficiently on GPUs due to frequent TLB misses, a high fraction of branch instructions, cache conflicts across cores, or irregular access strides to important data structures. We implement the selected kernels on GPUs using the CUDA programming model and use the PAPI CUDA component (Malony 2011) to measure and optimize performance. Initial results have identified the electron repulsion integrals, portions of the Hartree-Fock and MP2 computations, and the Becke grid weights computation as candidate kernels. Initial straightforward GPU implementations have achieved 4-17 times speedups compared to GAMESS on multicore CPUs. These results are consistent with by-hand GPU implementations that have already been carried out by the GAMESS team (Asadchev et al. 2010). We are working on further enhancement to optimize resource usage and increase the compute to memory access ratio.

## 5 CONCLUSIONS AND FUTURE WORK

We have identified the need for refactoring and automated performance tuning tools to facilitate adapting computational chemistry application codes to emerging complex hierarchical and heterogeneous high performance computing systems, including those with GPUs. We have begun working with the ROSE compiler infrastructure to implement refactoring of C and C++ codes and with the Eclipse Photran interface to implement refactoring of Fortran codes. We are using the SUPER autotuning framework to automate the process of identifying regions of code that are performance bottlenecks, outline these regions into separate kernels, and optimize performance of those kernels. We have begun using the PerfExpert and MACPO tools to identify regions that are amenable to being ported to GPUs. Initial results show signifi-

cant performance improvements and speedups with much less effort than would be required to refactor, port and tune the codes by hand.

The integral package that we have tuned for MPQC is just one of three integral computation packages supported by the MPQC super-integral object. We plan to integrate the two types of autotuning we have done to provide an autotuning framework for MPQC integral computations that encompasses both fine-grained loop transformations and coarse-grained parameter settings.

We have only scratched the surface on GPU implementations for portions of GAMESS. We plan to identify further kernels amenable to GPU implementation, use refactoring tools to make the kernels thread safe and outline them, and optimize the kernels for GPUs. We also plan to apply autotuning to the GAMESS+TigerCI code and parallelize the code for CPU+GPU systems.

## ACKNOWLEDGMENTS

This material is based upon work supported by the U.S. Department of Energy under Grant No. DE-FC02-06ER25761 and by the U.S. National Science Foundation under Grant No. CNS-0910899.

## REFERENCES

- Asadchev, A., V. Allada, J. Felder, B. M. Bode, M. S. Gordon, and T. L. Windus. 2010. "Uncontracted Rys Quadrature Implementation of up to G Functions on Graphical Processing Units." *Journal of Chemical Theory and Computation* 6 (3), 696-704.
- Bailey, D.H., Chame, J., Chen, C., Dongarra, J., Hall, M., Hollingsworth, J.K., Hovland, P., Moore, S., Seymour, K., Shin, J., Tiwari, A., Williams, S., You, H. 2008. "PERI Auto-tuning." Proc. SciDAC 2008, Journal of Physics, Seattle, Washington, Conference Series 125.
- Borkar S. and Chien, A.A. 2011. The future of microprocessors. *Commun. ACM* 54(5): 67-77.
- Browne, S., Dongarra, J., Garner, N., London, K., Mucci, P. 2000. "A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters," Proceedings of Super-Computing 2000 (SC'00), Dallas, TX, November 2000.
- Burtscher, M., B.D. Kim, J. Diamond, J. McCalpin, L. Koesterke, and J. Browne. 2010. "PerfExpert: An Easy-to-Use Performance Diagnosis Tool for HPC Applications." *SC 2010 International Conference for High-Performance Computing, Networking, Storage and Analysis*. November 2010
- Fowler, M. 1999. Refactoring: Improving the Design of Existing Code. Addison Wesley.
- Kenny, J.P. et al. 2012. "The Massively Parallel Quantum Chemistry Program (MPQC)." Sandia National Laboratories, Livermore, CA, USA, <http://www.mpqc.org/>
- Liao, C., D. J. Quinlan, R. Vuduc and T. Panas. 2009. "Effective Source-to-Source Outlining to Support Whole Program Empirical Optimization." 22nd International Workshop on Languages and Compilers for Parallel Computing, Newark, Delaware, USA. October 8-10, 2009.
- Lucas, R.F., et al. 2012. Institute for Sustained Performance, Energy, and Resilience (SUPER), <http://www.super-scidac.org/>.
- Malony, A., S. Biersdorff, S. Shende, H. Jagode, S. Tomov, G. Juckeland, R. Dietrich, D. Poole, and C. Lamb. 2011. "Parallel Performance Measurement of Heterogeneous Parallel Systems with GPUs," International Conference on Parallel Processing (ICPP'11), Taipei, Taiwan, September 13-16, 2011.
- Negara, S., G. Zheng, K.-C. Pan, N. Negara, R. E. Johnson, L V. Kale and P M. Ricker. 2010. "Automatic MPI to AMPI Program Transformation using Photran." 3rd Workshop on Productivity and Performance (PROPER 2010), Naples, Italy, August 31, 2010.
- Nielsen, I.M.B., and C. L. Janssen. 2007. "Local Møller-Plesset Perturbation Theory: A Massively Parallel Algorithm." *Journal of Chemistry and Theoretical Computing*, vol. 3 (2007), pg. 71.
- Nielsen, I.M.B., and C. L. Janssen. 2008. "Multicore Challenges and Benefits for High Performance Scientific Computing." *Scientific Programming*, vol. 16 (2008), pg. 277.



- Nieplocha, J., V. Tipparaju, M. Krishnan, and D. Panda. 2006. High Performance Remote Memory Access Communications: The ARMCI Approach. *International Journal of High Performance Computing and Applications* 20(2), 233-253.
- Oyeyemi, V.B., M. Pavone and E. A. Carter. 2011. "Accurate Bond Energies of Hydrocarbons from Complete Basis Set Extrapolated Multi-Reference Singles and Doubles Configuration Interaction". *Chem. Phys. Chem.*, **12**, 3354.
- Quinlan, D.J., et al. 2012. ROSE compiler project. <http://www.rosecompiler.org/>
- Rane, R. and J. Browne. 2011. "Performance Optimization of Data Structures Using Memory Access Characterization." *CLUSTER 2011*: 570-574
- Schmidt, M.W., K. K. Baldridge, J. A. Boatz, S. T. Elbert, M. S. Gordon, J. H. Jensen, S. Koseki, N. Matsunaga, K. A. Nguyen, S. Su, T. L. Windus, M. Dupuis, and J. A. Montgomery, Jr. 1993. "The General Atomic and Molecular Electronic Structure System." *Journal of Computational Chemistry*, vol. 14 (1993), pg.1347.
- Seymour, K., You, H., and Dongarra, J. 2008. "A Comparison of Search Heuristics for Empirical Code Optimization," *The 3rd international Workshop on Automatic Performance Tuning*, Tsukuba, Japan, October 1, 2008.
- Valeev, E. F., and C. L. Janssen. 2004. "Second-Order Møller-Plesset Theory with Linear R12 Terms (MP2-R12) Revisited: Auxiliary Basis Set Method and Massively Parallel Implementation." *Journal of Chemical Physics*, vol. 121 (2004), pg. 1214.
- Watson, G. and N. Debardeleben. 2006. "Developing scientific applications using Eclipse," *Computing in Science and Engineering* 8(4): 50-61.

## AUTHOR BIOGRAPHY

**SHIRLEY MOORE** is an Associate Professor in the Computer Science Department and graduate Computational Science Program at the University of Texas at El Paso. Her research interests are in performance modeling and optimization of parallel scientific applications and in applying software engineering methodologies to high performance computing applications. Her email address is [svmoore@utep.edu](mailto:svmoore@utep.edu).