Multilevel Parallelism in Computational Chemistry using Common Component Architecture and Global Arrays

Manojkumar Krishnan Yuri Alexeev Theresa L. Windus Jarek Nieplocha
Pacific Northwest National Laboratory
Richland, WA 99352, USA.
{manoj, yuri.alexeev, theresa.windus, jarek.nieplocha}@pnl.gov

ABSTRACT
The development of complex scientific applications for high-end systems is a challenging task. Addressing complexity of the involved software and algorithms is becoming increasingly difficult and requires appropriate software engineering approaches to address interoperability, maintenance, and software composition challenges. At the same time, the requirements for performance and scalability to thousand processor configurations magnifies the level of difficulties facing the scientific programmer due to the variable levels of parallelism available in different algorithms or functional modules of the application. This paper demonstrates how the Common Component Architecture (CCA) and Global Arrays (GA) can be used in context of computational chemistry to express and manage multi-level parallelism through the use of processor groups. For example, the numerical Hessian calculation using three levels of parallelism in NWChem computational chemistry package outperformed the original version of the NWChem code based on single level parallelism by a factor of 90% when running on 256 processors.

Categories and Subject Descriptors
D.1.3 [**Programming Techniques**]: Concurrent Programming – *Parallel programming*.

General Terms
Algorithms, Performance, Design (c) 2005 Association for Computing Machinery.

Keywords
Multilevel parallelism, Component, Global Arrays, Parallel programming, Processor groups.

1. INTRODUCTION
Exploiting all available forms of parallelism is becoming increasingly important for programming forthcoming high-end systems. Such systems, containing tens or hundreds of thousands of processors, present a challenge to many important scientific applications. Many of the applications that require high-end systems tend to be composed of algorithms with variable computation/communication granularity. The question on how to partition computational resources and manage them to execute the overall application effectively is becoming critical for our ability to take advantage of the massively parallel hardware. One strategy to limit the negative effect of Amdahl's law on the overall efficiency and scalability of the application is to execute the finer granularity algorithms on smaller subsets of processors, where their efficiency and speedup are high.

Common Component Architecture [1, 2] has been proposed as a technology for building complex scientific applications as a collection of reusable components that encapsulate the required fundamental algorithms, solvers, and methods. These components are designed from scratch or adopted from existing applications or libraries to form reusable building blocks with standardized interfaces. The main motivation is to be able to reuse and swap components as needed with minimum effort.

This paper describes how CCA and GA processor groups can be deployed together to manage multilevel parallelism in computational chemistry algorithms. This effort has been pursued in the context of NWChem [3], a large (2.5million lines of code) software suite that encompasses multiple theories, algorithms, and methods in the molecular computational chemistry domain. NWChem was developed using multiple programming languages (Fortran, C, C++, and Python) and programming models (MPI, Global Arrays). Although NWChem has been designed from scratch to work on massively parallel systems, until now it was unable to effectively exploit variable degrees of parallelism available in the set of algorithms and methods it offers. As a result, the scalability of some important calculations was limited by the least scalable parts of the simulation. To address this scalability limitation, it was necessary to add group awareness in the Global Arrays toolkit [4], in addition to CCA and the processor group management MPI offers. The Global Arrays [5, 6], shared memory programming toolkit, has been used by NWChem as the primary programming model and became the enabling technology for rapid and scalable implementation of algorithms in this application area.

A large number of computational chemistry methods such as numerical Hessian, simulated annealing and global optimization methods, fragment molecular orbital method (FMO) [7], and vibrational self-consistent field can benefit from decomposition of the workload at variable levels of granularity. The common element for all these methods is numerous computations of limited scalable single operation (energy or gradient). The

SCl05 November 12-18, 2005, Seattle, Washington, USA.

efficient management of CPUs can significantly improve performance. In our paper, this is demonstrated for a numerical Hessian method. Numerical Hessian is computed from numerical gradients which in turn are computed numerically from energy calculations. Each energy calculation has limited scalability and may require multiple nodes due to memory requirements. Thus to compute numerical gradients, subgroups need to be created. There is always an optimum number of CPUs in the group which defines optimum number of CPUs that should be allocated for a single numerical gradient calculation. The proper allocation considers computational efficiency as well as available memory. If chosen incorrectly, the allocation may lead to workload imbalance and overall poor performance.

This paper makes several contributions. First it describes how CCA can be deployed in the context of multi-level parallelism on a high-performance computer. To the best of our knowledge, this is the first work reporting usage of CCA in this context. Second, the paper discusses how this technology can be used to exploit variable concurrency in the computational chemistry area. Third the paper validates the overall strategy by demonstrating performance improvements for NWChem Hessian calculations. For example, numerical Hessian calculation using three levels of parallelism outperformed the original version of the code based on one level of parallelism by a factor of 90% when running on 256 processors.

2. COMMON COMPONENT ARCHITECTURE

The Common Component Architecture (CCA) provides means for scientific programmers to manage the complexity of large-scale scientific simulations and to move toward a *plug-and-play* environment for high-performance computing [2]. The CCA is a component model specifically designed to address the needs of high-performance scientific computing by the CCA Forum [1]. From an application scientist's perspective, components allow software developers to describe the calling interfaces of libraries and applications in a manner that hides low-level details, such as implementation language, compiler, parallelism, or location on a network. Components encapsulate the knowledge, experience, and work of other scientists, and they provide building blocks that speedup application development. To exploit these benefits in applications, we have developed CCA component interfaces for NWChem [3].

While the details of the CCA specification [8] are beyond the scope of this paper, we highlight the key points that are most pertinent for the combined parallel use of NWChem components in the multiple-program multiple-data (MPMD) mode as well as in the single-program multiple-data (SPMD) mode [2, 9, 10]. The CCA approach consists of three main elements: *components*, *ports*, and *frameworks*. Briefly, *components* are basic units of software functionality that can be composed together at runtime to form applications, while *ports* are the abstract interfaces through which components interact. *Frameworks* manage components as they are assembled into applications and executed. One of the fundamental assumptions in the CCA is that components may be written in different programming languages. To facilitate language interoperability, the Scientific Interface Definition Language (SIDL) from Babel [11, 12] has been broadly adopted to describe CCA interfaces of scientific components, including NWChem component.

Babel is an interface definition language (IDL)-based tool that automatically generates code to glue multi-language components together. It relies on the Scientific Interface Definition Language (SIDL) [11] for the definitions of calling interfaces through defined types (i.e., interfaces and classes) and declared methods. Since component technologies are an evolutionary step beyond object-oriented programming, the CCA has been able to leverage Babel and SIDL in the development of its component framework [2]. The CCA specification is written in SIDL, and component developers write SIDL files to describe their ports and the classes that use or provide those ports. Using SIDL enables the encapsulation of implementation details of CCA-compliant components.

2.1 MCMD Driver Architecture

Ccaffeine [13], the main CCA framework implementation for high performance computing, supports both the single program/multiple data (SPMD) and multiple program/multiple data (MPMD) models. In the context of CCA, we refer these models as single or multiple *component*/multiple data (SCMD, MCMD) models [2].

In the CCA framework, by default applications run as single program, multi-component applications, where each component is loaded across all of the processors. However, to create the dynamic environment to improve application efficiency and manage the resources effectively, we will be instantiating components on subgroups of processors. In CCA, this means that the *BuilderService* capability must be exploited to its fullest. *BuilderServices* in CCA provide means to programmatically assemble and modify applications (instantiate and destroy components), and means for an arbitrary code to become a part of the CCA framework [2]. These services facilitate dynamic behavior of the application itself, for example, swapping components based on numerical or computational performance [14]. *BuilderService* also enables encapsulation of groups of components so that they can be treated as a single component. This enables effective management of large component-based applications, including the assembly of multi-scale or multi-physics simulations, where complex applications representing a particular length scale or type of physics can be encapsulated and treated as a single component while exposing only a limited number of ports.

In most existing CCA applications, user normally starts up a framework and then instantiates components. However, in the MCMD case, to accomplish the dynamic creation and destruction of components, we need to create and start an MCMD driver. The driver starts the framework as a component and then attaches other components to that framework as needed. Some components are typically loaded into all processes, while others are loaded only into subsets of processes. These separate components are managed by the MCMD driver component and interfaced by other components, which for example handle the data exchange. Because of the inherent complexity of partitioning the process space and of launching parallel jobs with different inputs or executables on every process, MCMD applications can be most easily created using the CCA's *BuilderService* from a so-called builder component. This component would compute the desired partition of the available set of processes and then use

BuilderService on each process to load and connect the appropriate set of components. The builder would then invoke the *GoPort* on the MCMD driver component to initiate the simulation.

3. PROCESSOR GROUPS AND GA PROGRAMMING MODEL

The Global Array (GA) toolkit provides a high-level programming model that offers a shared memory style communication in context of distributed data structures. NWChem and several other computational chemistry packages rely on GA as the underlying programming model. Processes can communicate with each other by creating and accessing GA distributed matrices as well as conventional message-passing (MPI). GA is compatible with MPI and allows full interoperability with software such as numerical linear algebra libraries developed on top of MPI. Each process can independently and asynchronously access any two-dimensional patch of a GA distributed matrix, without requiring cooperation by the application code in any other process. Each process is assumed to have fast access to some portion of each distributed matrix, and slower access to the remainder. These speed differences define the data as being 'local' or 'remote', respectively. If the data is 'local', a process can directly access the memory block to retrieve data instead of using 'get' access. The GA toolkit offers support for both *task* and *data* parallelism. *Task parallelism* is supported through the one-sided (noncollective) copy operations that transfer data between global memory (distributed/shared array) and local memory. In addition, each process is able to access directly data held in a section of a global array that is logically assigned to that process. The one-sided communications used by Global Arrays eliminate the need for the programmer to account for responses by remote processors. The data parallel computing model is supported through the set of collectively called functions that operate on either entire arrays or sections of global arrays. The set includes BLAS-like operations (copy, additions, transpose, dot products, matrix multiplication, etc,). These are collective data-parallel operations that are called by all processes in the parallel job.

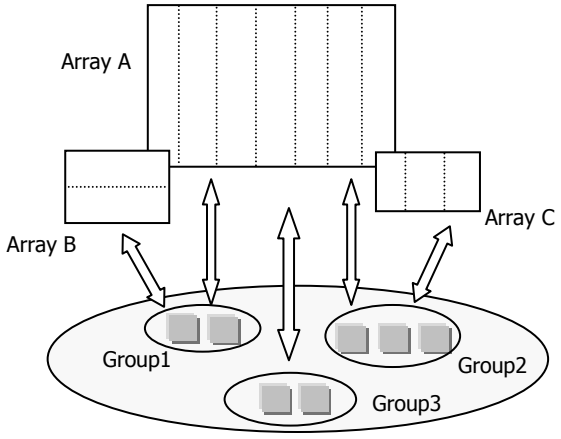

Figure 1. An example of multilevel parallelism in Global Arrays.

The development of multi-level parallel algorithms in NWChem has been enabled by introduction of the processor group support in GA [4]. Due to the required compatibility of GA with MPI, the MPI approach to the processor group management was followed as closely as possible. However, in shared memory programming, management of memory and shared data rather than management of processor groups itself is the primary focus. More specifically we need to determine how to create, efficiently access, update, and destroy shared data in the context of the processor management capabilities that MPI already provides. For example, Figure 1 illustrates the concept of using shared arrays. The three processor groups (Group1, Group2, and Group3 in Figure 1) execute tasks that operate on three arrays: A, B, and C. Array A is in the scope of all three processor groups. Array B is distributed on processor Group 1. Array C is distributed on processor group 3. All arrays can be accessed using collective (individual and multiple arrays) and one-sided (non-collective) operations.

One of the fundamental group-aware GA operations involves the ability to create shared arrays on subsets of processors. Every global array has only one associated processor group specifying the group that created the array. Another useful operation is the data-parallel copy operation that works on arrays (or subsections) defined on different processor groups as long as the intersection of these groups is a non-empty set. In the GA programming model, data distributed in a processor group (containing *M* processors) can be redistributed to another processor group (containing *N* processors) regardless of the number of processors in each group and the data layout. This can be done as a collective call across processors in both the groups or as a non-collective one-sided operation. This feature enabled development of applications with nontrivial relationships between processor groups.

The concept of the *default processor group* is a powerful capability added to enable rapid development of new group-based codes and simplify conversion of the existing non-group aware codes. Under normal circumstances, the default group for a parallel calculation is the MPI "world group" (contains the complete set of processors user allocated). However, in GA, a call is available that can be used to change the default group to a processor subgroup. This call must be executed by all processors in the subgroup. Once the default group has been set, all operations are implicitly assumed to occur on the default processor group unless explicitly stated otherwise. By default, GA shared arrays are created on the default processor group and global operations by default are restricted to the default group. Inquiry functions, such as the number of nodes and the node ID, return values relative to the default processor group.

4. NWCHEM CCA COMPONENT

Our implementation of the model class contains a molecule class, which maintains molecular structure information, and method implementations for molecular energy and energy derivative evaluations. All methods use Cartesian coordinates in the arguments and return values for chemistry models. Provided the chemistry models return quantities in the same reference frame as the given coordinates, the details of symmetry implementation and molecule reorientation will not cause conflicts between model implementations. Although the specific implementation of a model varies based on the design of the particular quantum

chemistry package used, the initialization of a model generally requires the collection of input parameters such as the level of theory, choice of atomic orbital basis set, and molecular structure. An implementation of the *ModelFactory interface* is provided to collect input parameters and provide them to the initialization member of the model class, returning the initialized model via its port.

NWChem is a computational chemistry code created for massively parallel computations. It has many theoretical and algorithmic methods available for electronic structure calculations with Gaussian and plane-wave basis sets as well as classical molecular dynamics capabilities. Although NWChem uses object-oriented principles, it is implemented in a combination of Fortran, C, and Python (with Python used mostly for prototyping capabilities). Therefore, the CCA *Model Factory* instantiates essentially the same NWChem object for all initializations because it is not "true" C++ object. For our purposes it does not restrict our implementation in any way.

Our first implementation of the NWChem Model Factory and model functionality used C++ and C++ wrappers because the CCA Ccaffeine framework [13] initially worked only with C++ components. However, the latest versions of the framework are Babel aware, and allow the component developer to use languages supported by Babel. We have now implemented the NWChem model functionality with native Fortran, which makes our development much more straightforward. This approach will also enable other components that rely on lower level routines to be implemented easily.

Implementation of the Model and ModelFactory interfaces provides the basic functionality of quantum chemical codes [15]. The paradigm followed here is to build higher level application-specific component systems on top of these core chemistry components, using existing general-purpose components when possible.

5. EXAMPLE APPLICATION USING MULTI-LEVEL PARALLELISM

In this section, we describe the multi-level parallelism scheme that may be applied to perform numerical Hessian, simulated annealing, and other methods. Our previous work [15] on components technology involved two quantum chemistry packages, Massively Parallel Quantum Chemistry (MPQC) [16, 17], NWChem [3] and Global Arrays (GA) [6, 18]. All of these packages are designed for high performance on parallel hardware.

5.1 Numerical Hessian Architecture

Numerical Hessians [19, 20] are perhaps the simplest example of what can be accomplished with CCA architecture. We intentionally chose this system to demonstrate the power of multilevel parallelism. However, the multilevel parallel scheme used in this example also opens the door to a rich set of algorithms that require a very dynamic architecture to enable advancement in large scale chemical simulations. In the overall algorithm, a CCA MCMD Hessian driver component is used to instantiate NWChem Quantum Mechanics (QM) components on subsets of processors to calculate gradients. These components in turn, may need to calculate numerical gradients and will therefore, need to create subgroups to calculate multiple energies. This particular algorithm offers essentially three levels of parallelism as shown in Figure 2: one at the CCA (Hessian) level, one at the gradient level and another to calculate the energy (each energy itself can use a large number of processors).

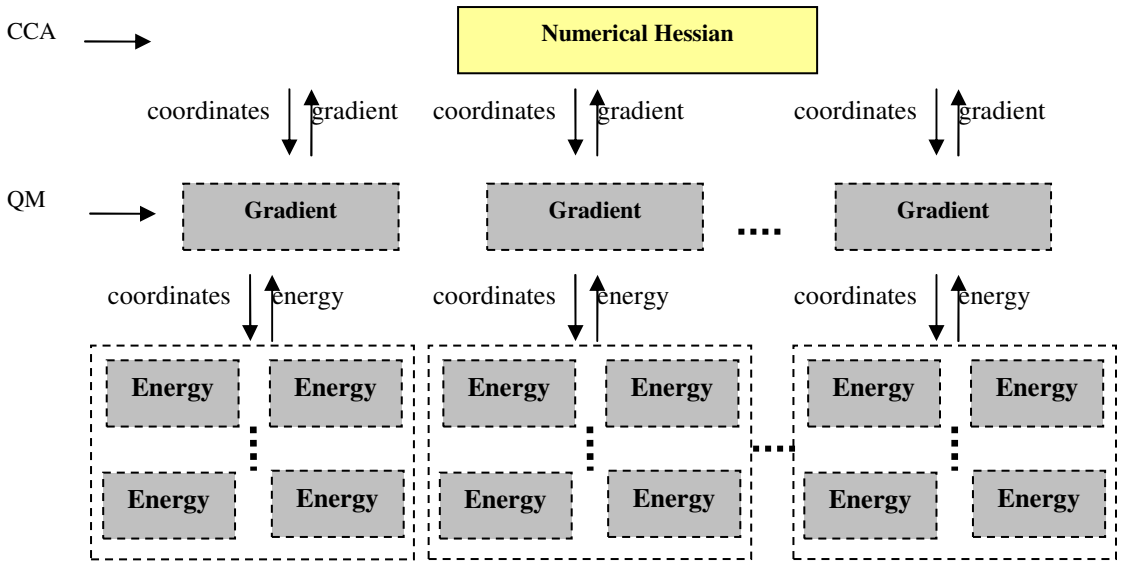

Figure 2. Example of three level parallelism in numerical Hessian computations. The top layer involves the CCA framework and the lower layer involves subgroup capability within NWChem and GA.

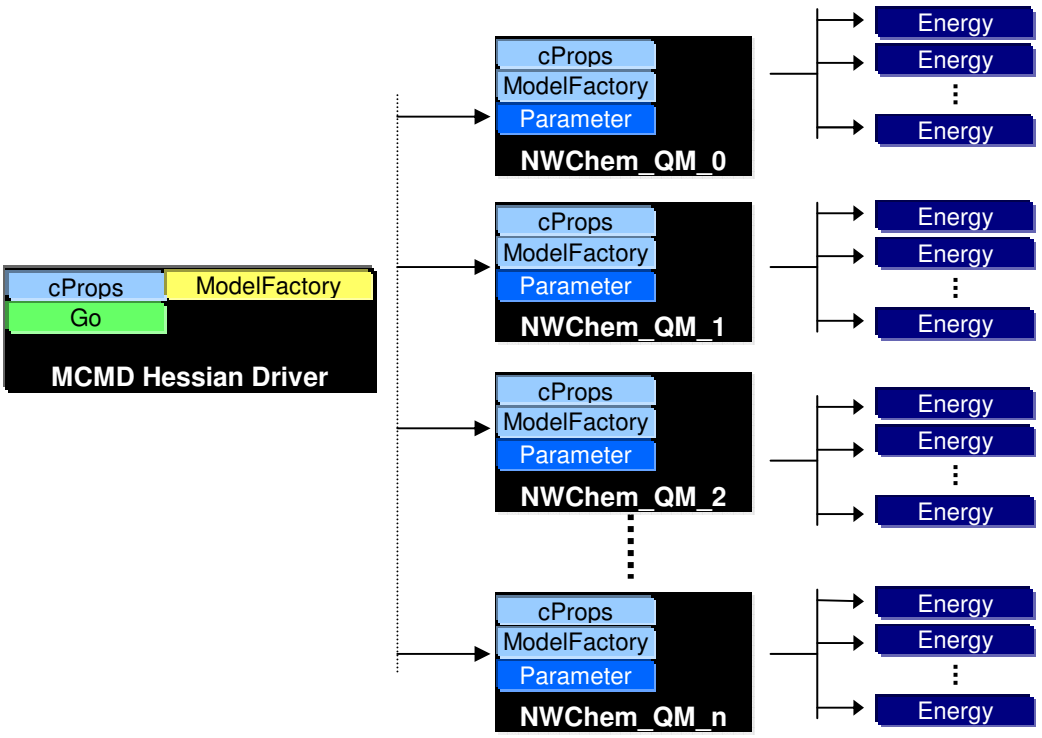

Figure 3. MCMD Driver launches multiple instances of NWChem QM components on subsets of processors (also assigns a GA communicator for every instance). Each NWChem QM component does multiple energy computations on subgroups.

In our case, a driver Hessian component is created which then uses the CCA BuilderServices to start the framework, to create multiple processor subgroups and then create different instantiations of QM ModelFactory component on processor subgroups via the provided ModelFactory port, to calculate gradients as shown in Figure 2. ModelFactory instantiates a Model class which is provided in model options (NWChem in our case). One of the Model class methods is used to compute numerical gradient. The Model is configured via CCA's parameter port. The input parameters are theory, basis set, and NWChem input file where the Cartesian coordinates and startup NWChem options for numerical gradient are set.

NWChem relies on GA for its underlying parallel programming model, and subgroups are formed in the Hessian MCMD driver by relying on the *default group* capability described in Section 3. This capability of GA allowed the components to be instantiated and then operate in a more transparent fashion than in the case of MPI which requires explicit specification of the corresponding processor group communicator in all the communication interfaces. In particular, the development of this component due to the implicit nature of processor groups required virtually no changes to the original computational code in NWChem (except for changes to the infrastructure described below) that was designed to exploit only one level of parallelism.

Each of the QM components then needs to compute a gradient. This requires multiple energy computations which can efficiently run on subgroups. Here again, groups are formed and the energies are computed in a task parallel fashion as shown in Figure 3. In addition to the GA capabilities, additional modifications are required within NWChem parallel infrastructure to facilitate their use. Cloning of local information (e.g. runtime database, molecular orbitals) is necessary so that each subgroup has all of the required information to accomplish the required task (in this case an energy evaluation). Task parallelism and load balancing in the gradient level is required to obtain an even workload distribution. This type of load balancing is also required in the Hessian driver component.

With the CCA's MCMD model, our Hessian MCMD driver manages all of the available parallel processors dynamically by instantiating and connecting infrastructure components based on application logic. Multiple QM Components of the NWChem model are run simultaneously on different subsets of the available processes. These separate elements are managed by the MCMD driver component and supported by GA to handle the data exchange between the various elements. Because of the complexity of partitioning up the process space and of launching parallel jobs with different inputs or executables on every process, numerical Hessians can perhaps be created through the use of the CCA's *BuilderService* framework service from an MCMD driver component. This component would compute the desired partition of the available processes and instantiate multiple QM components based on the number of gradient and energy calculations.

It should be noted here that there is no restriction that requires each of the subgroups at a particular level to perform the same work as in this example. This architecture allows for maximum

flexibility in the high performance algorithm and some processor groups could be computing gradients while others could be doing optimizations or other tasks. Of course load balance would be a critical component of any such algorithm.

6. EXPERIMENTAL RESULTS

Water clusters form local networks which exhibit large cooperative effects. Infrared (IR) spectroscopy can provide structural information of the network and its connectivity. If the spectroscopic signatures (positions and intensities of IR bands) of the different hydrogen bonding networks are known, then the experimental spectra can be directly related to a structure [21]. Quantum chemistry calculations can provide the necessary IR spectra of the local networks to link "structural-spectral" correspondence and is accomplished by computing Hessians (from which IR frequencies can be easily obtained) for different size water clusters. The Hessians of water clusters up to 20 water molecules have been computed by Xantheas et al [22]. The proposed scheme will allow researchers to compute Hessians of significantly larger clusters by efficiently utilizing large number of CPUs.

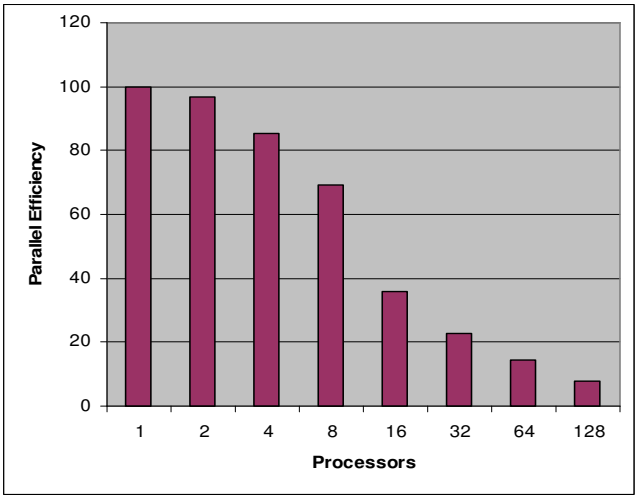

Figure 4. Parallel efficiency of single energy calculation with single-level parallelism.

In the case study we will demonstrate the proposed approach on computing numerical Hessian of $(H_20)_5$. The calculations have been done at the MP2/cc-pVDZ level of theory on already optimized $(H_20)_5$ [23]. First and second derivatives of the energy with respect to nuclear displacements were computed with the NWChem component. The calculations were performed on the massively parallel Linux cluster (dual-node 1.5 GHz Itanium with Quadrics QsNet-II interconnect) at the Molecular Science Computing Facility in the William R. Wiley Environmental Molecular Sciences Laboratory at Pacific Northwest National Laboratory.

The scalability of single energy calculation is shown on Figure 4. The most efficient calculation of course is on one or two CPUs but due to large memory requirements the optimal number of CPUs is 4. If more than 4 CPUs are allocated to compute the Hessian then it is inefficient. One way to address this problem is the introduction of subgroups. The computation of a single gradient calculation is performed via round robin over multiple energy calculations. In the case of $(H_20)_5$ there are 39 steps (positive and negative offsets to the coordinate per step) with a total of 78 energy calculations per gradient. Total number of steps corresponds to the total number of symmetrically unique modes. It is obvious that the number of subgroups allocated for one gradient calculation should be less than 39 and divisible by the number of subgroups. If not, there is workload imbalance which is accumulated over the total number of gradient calculations. Moreover, load balancing in the gradient level is required to obtain an even workload distribution. Thus, this application presents opportunities to decompose the workload at variable levels of granularity. It is also difficult to manage all of the available processors dynamically in the most efficient manner possible.

To demonstrate the benefits of multi-level parallelism in NWChem using CCA and GA, we conducted experiments to compute the numerical Hessian in three different ways:

- without groups. i.e. the native parallel code (*native* method)
- with processor subgroups, each group having 4 processes. (*native-groups* method)
- using MCMD multilevel parallelism (*mcmd* method).

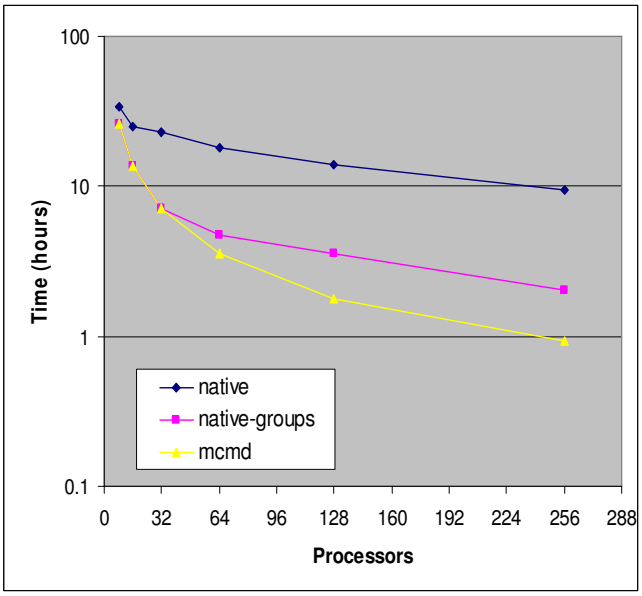

Figure 5. Scalability of numerical Hessian calculation.

The scalability of numerical Hessian is shown in Figure 5. The *native* way of computing numerical Hessian is also not scaling well because the single energy calculation is not scaling beyond 4 processors (see Figure 4). A better scaling is achieved in the case of *native-groups* method, as processor subgroups (4 processes in a group) are used for multiple energy calculations to compute a gradient. However, there is a potential chance for load imbalance. Consider the case of 128 processors where there are 32 processor subgroups calculating 39 steps per gradient. Scaling is affected due to coarse load imbalance, as the number of energy calculations is not divisible by the number of subgroups. Some of subgroups will compute 1 step and others will compute 2 steps which give 64% imbalance. The imbalance will increase with number of processors as shown in Figure 6.

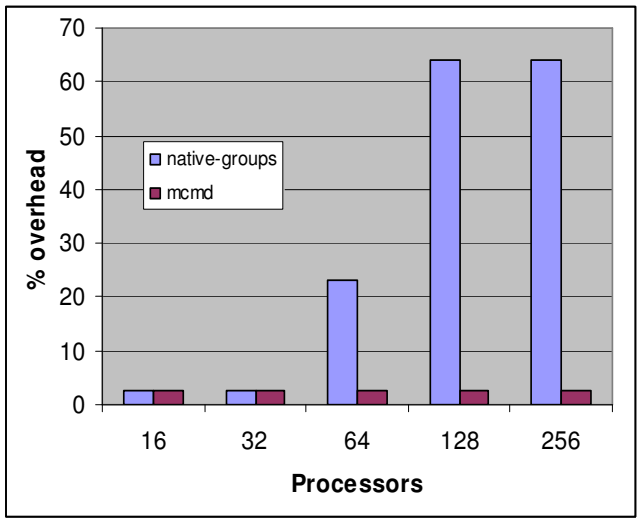

Figure 6. Load imbalance overhead at the gradient level.

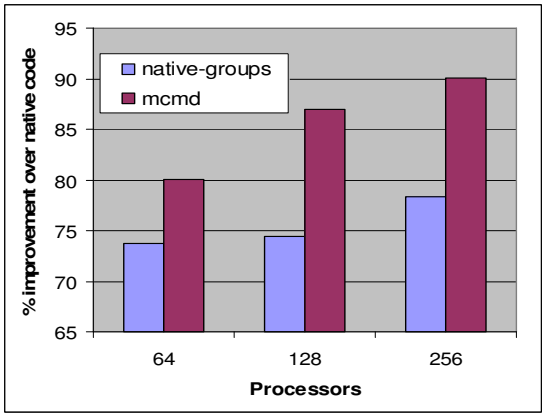

Figure 7. NWChem numerical Hessian computations. Percentage improvement of *native-group* and *mcmd* approach performance with respect to the native approach.

Thanks to the CCA MCMD model, multiple components (task/program) can be launched on subsets of processors. Thus, task parallelism and load balancing in the gradient level are addressed to obtain fine load distribution in this case. The MCMD driver dynamically instantiates components based on the number of energy calculations per gradient. As we see from Figure 6, load balancing is a serious problem beyond 32 processors, and therefore, the MCMD driver instantiates multiple components for larger processor runs (> 32 processors). Moreover, collective operations scale well as we have multiple NWChem instances running on subsets of processors, rather than a single instance of the NWChem application running on all the processors. Figure 7 shows the performance improvement of *mcmd* and *native-groups* approach with respect to the *native* method.

Based on the presented results, we recommend the following methodology to achieve optimal performance for a particular problem. First step is computing scalability of single energy calculation. The optimal number of processors will define number of processors in a subgroup for each energy calculation. The second step is to assign optimal number of subgroups per gradient calculation. This number can be estimated based on total number of steps needed to compute single gradient. The acceptable imbalance should be less than 10%. The future generation of numerical Hessian will be capable of deriving these parameters automatically at all levels.

7. CONCLUSIONS

This work has shown a novel usage of CCA technology and subgroups within GA programming model to implement flexible, multi-level software architecture for computational chemistry applications on high performance computers. This approach aims to better exploit variable level of parallelism in large complex applications. The experimental results demonstrated very impressive improvements in scalability by taking advantage of multiple levels of parallelism. In particular, the numerical Hessian calculation using three levels of parallelism outperformed the original version available in the NWChem package based on single level parallelism by a factor of 90% when running on 256 processors. The MCMD Hessian driver used the processor resources effectively for this problem and obtained results in less than hour, compared to native code which took ten hours on 256 processors.

While only a limited number of processors were used in this work, the proposed approach is directly applicable to systems with much larger processor counts. Based on proposed methodology it should be also straightforward to adopt other CCA components that are processor-group aware to implement other types of scientific computations, for example involving optimization and linear algebra.

8. ACKNOWLEDGMENTS

We thank our colleagues within the DoE SciDAC Center for Component Technology for Terascale Simulation Software (CCTTSS) for stimulating discussions of issues in component and interface design. We would also like to specially thank the CCTTSS members Wael Elwasif, Ben Allan, Rob Armstrong and Joseph Kenny for their advice and discussions in this project.

The research described in this paper was supported by the Department of Energy, Office of Advanced Scientific Computing Research at the Pacific Northwest National Laboratory, a multiprogram national laboratory operated by Battelle for the U.S. Department of Energy under Contract DE-AC06-76RL01830. This research was performed in part using the Molecular Science Computing Facility (MSCF) in the William R. Wiley Environmental Molecular Sciences Laboratory, a national scientific user facility sponsored by the U.S. Department of Energy's Office of Biological and Environmental Research and located at the Pacific Northwest National Laboratory. Pacific Northwest is operated for the Department of Energy by Battelle.

9. REFERENCES

[1] CCA-Forum. Common Component Architecture Forum. http://www.cca-forum.org.

[2] D. E. Bernholdt, B. A. Allan, R. Armstrong, F. Bertrand, K. Chiu, T. L. Dahlgren, K. Damevski, W. R. Elwasif, T. G. W. Epperly, M. Govindaraju, D. S. Katz, L. F. Diachin, J. A. Kohl, M. Krishnan, G. Kumfert, S. Lefantzi, M. J. Lewis, A. D. Malony, L. C. McInnes, J. Nieplocha, B.

Norris, S. G. Parker, J. Ray, S. Shende, T. L. Windus, and Zhou.S., "A Component Architecture for High-Performance Scientific Computing," Intl. J. High-Perf. Computing Appl., 2004.

[3] R. A. Kendall, E. Apra, D. E. Bernholdt, E. J. Bylaska, M. Dupuis, G. I. Fann, R. J. Harrison, J. L. Ju, J. A. Nichols, J. Nieplocha, T. P. Straatsma, T. L. Windus, and A. T. Wong, "High performance computational chemistry: An overview of NWChem a distributed parallel application," Computer Physics Communications, vol. 128, pp. 260-283, 2000.

[4] J. Nieplocha, M. Krishnan, B. Palmer, V. Tipparaju, and Y. Zhang, "Exploiting Processor Groups to Extend Scalability of the GA Shared Memory Programming Model," in proceedings of ACM Computing Frontiers, Italy, 2005.

[5] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global Arrays: A Portable Shared Memory Programming Model for Distributed Memory Computers," in proceedings of Supercomputing, 1994.

[6] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apra, "Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit," International Journal of High Performance Computing Applications, 2005.

[7] D. G. Fedorov, R. M. Olson, K. Kitaura, M. S. Gordon, and S. Koseki, "A new hierarchical parallelization scheme: Generalized distributed data interface (GDDI), and an application to the fragment molecular orbital method (FMO)," Journal of Computational Chemistry, vol. 25, pp. 872 - 880, 2004.

[8] B. A. Allan, R. C. Armstrong, A. P. Wolfe, J. Ray, D. E. Bernholdt, and J. A. Kohl, "The CCA Core Specification in a Distributed Memory SPMD Framework," Concurrency Computataions, vol. 14, pp. 1-23, 2002.

[9] L. C. McInnes, B. A. Allan, R. Armstrong, S. J. Benson, D. E. Bernholdt, T. L. Dahlgren, L. F. Diachin, M. Krishnan, J. A. Kohl, J. W. Larson, S. Lefantzi, J. Nieplocha, B. Norris, S. G. Parker, J. Ray, and Zhou.S., "Parallel PDE-Based Simulations Using the Common Component Architecture," Argonne National Laboratory ANL/MCS-P1179-0704. To appear as an invited chapter in the book Numerical Solution of Partial Differential Equations on Parallel Computers, A. M. Bruaset, P. Bjorstad, and A. Tveito, editors, Springer-Verlag, 2004.

[10] CCA-Spec. http://www.cca-forum.org/docs/specification.html.

[11] T. Dahlgren, T. Epperly, and G. Kumfert, "Babel User's Guide," Lawrence Livermore National Laboratory 2004.

[12] T. Dahlgren, T. Epperly, and G. Kumfert, "Babel/SIDL Design-by-Contract: Status," Lawrence Livermore National Laboratory UCRLPRES-152674, 2003.

[13] B. Allan, R. Armstrong, S. Lefantzi, J. Ray, E. Walsh, and P. Wolfe. Ccaffeine - a CCA Component Framework for Parallel Computing. http://www.cca-forum.org/ccafe/.

[14] D. E. Bernholdt, R. C. Armstrong, and B. A. Allan, "Managing complexity in modern high end scientic computing through component-based software engineering," in proceedings of HPCA Workshop on Productivity and Performance in High-End Computing (P-PHEC 2004), Madrid, Spain., 2004.

[15] J. P. Kenny, S. J. Benson, Y. Alexeev, J. Sarich, C. L. Janssen, L. C. McInnes, M. Krishnan, J. Nieplocha, E. Jurrus, C. Fahlstrom, and T. L. Windus, "Component-based integration of chemistry and optimization software," Journal of Computational Chemistry, vol. 25, pp. 1717-1725, 2004.

[16] C. Jansen. The Massively Parallel Quantum Chemistry Program. http://aros.ca.sandia.gov/~cljanss/mpqc/index.html.

[17] C. L. Janssen, E. T. Seidl, and M. E. Colvin, "Object-oriented implementation of a Parallel Ab-initio Program," Parallel Computing in Computational Chemistry ACS Symposium Series, American Chemical Society, Washington,DC, 1995, vol. 592, pp. 47, 1995.

[18] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global arrays: A nonuniform memory access programming model for high-performance computers," Journal of Supercomputing, vol. 10, pp. 169-189, 1996.

[19] T. L. Windus, M. W. Schmidt, and M. S. Gordon, "Parallel algorithm for SCF analytic Hessians. I. Small scale algorithm," Chemical Physics Letters, vol. vol 216, pp. 375-379, 1993.

[20] A. M. Marquez, J. Oviedo, J. F. Sanz, and M. Dupuis, "Parallel Computation of second derivatives of RHF rnergy on distributed memory computers," Journal of Computational Chemistry, vol. 18, pp. 159-168, 1997.

[21] T. S. Zwier, "The Structure of Protonated Water Clusters," Science 304, pp. 119-120, 2004.

[22] G. S. Fanourgakis, A. Aprà, W. E. de Jong, and S. S. Xantheas, "High-level ab initio calculations for the four low-lying families of minima of (H2O)20. II. Spectroscopic signatures of the dodecahedron, fused cubes, face-sharing pentagonal prisms, and edge-sharing pentagonal prisms hydrogen bonding networks," Journal of Chemical Physics, vol. 122, pp. 134304-13, 2005.

[23] S. S. Xantheas, "Ab initio studies of cyclic water clusters (H2O)n, n=1-6. III. Comparison of density functional with MP2 results," The Journal of Chemical Physics, vol. 102, pp. 4505-4517, 1995.