

# Raising the Level of Programming Abstraction in Scalable Programming Models

David E. Bernholdt  
*Computer Science & Mathematics*  
*Oak Ridge National Laboratory*  
bernholdtde@ornl.gov

Jarek Nieplocha  
*Computational Sciences & Mathematics*  
*Pacific Northwest National Laboratory*  
j\_nieplocha@pnl.gov

P. Sadayappan  
*Dept. of Computer and Information Science*  
*Ohio State University*  
saday@cis.ohio-state.edu

## Abstract

*The complexity of modern scientific simulations combined with the complexity of the high-performance computer hardware on which they run place an ever-increasing burden on scientific software developers, with clear impacts on both productivity and performance. We argue that raising the level of abstraction of the programming model/environment is a key element of addressing this situation. We present examples of two distinctly different approaches to raising the level of abstraction of the programming model while maintaining or increasing performance: the Tensor Contraction engine, a narrowly-focused domain specific language together with an optimizing compiler; and Extended Global Arrays, a programming framework that integrates programming models dealing with different layers of the memory/storage hierarchy using compiler analysis and code transformation techniques.*

## 1. Introduction

The role of computational simulation in science and engineering has blossomed in recent years to the point where it is now recognized as a peer to experimental and theoretical approaches and has become an indispensable tool to the progress of modern science and technology. Moreover, the pace of change and improvement in scientific high-end computing has been tremendous: more powerful computers allow researchers to perform larger and higher fidelity simulations, which in turn inspire the

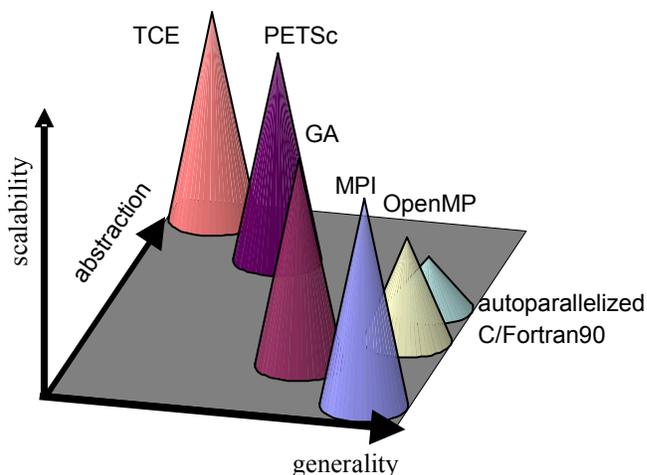
need for yet larger and faster computers. However this progress has not been without cost. Software developers have had to face increases in the complexity of algorithms and methods concomitant with the increases in problem size and fidelity compounded by increases in software complexity required to tease the maximum performance out of hardware with deeper memory hierarchies, higher degrees of parallelism, and other “features”. The result of this burgeoning complexity is that more and more of the software developer’s effort goes into dealing with the details, with obvious impacts on overall productivity.

Any measure of productivity for a developer and user of software must take into account both the time required to develop the software and the time it takes to run, or the performance. A “productive” programming environment, therefore, is one that allows the programmer to easily express computational problem (i.e. a programming model which provides a high level of abstraction) while providing the highest possible performance. Based on our collective experience in high-performance scientific computing and our assessment of progress in the field over the last 10-15 years, we argue that raising the level of abstraction available to the developer has become a crucial factor in the effort to increase software productivity in scientific computing. After discussing the idea of abstraction in high-end computing, this paper presents the Tensor Contraction Engine (TCE) and Extended Global Arrays (XGA) as examples of efforts that take different approaches toward the goal of raising the level of abstraction while maintaining high performance.

## 2. Abstraction, Scalability, and Generality in High Performance Programming

For the programming model on which the software development effort is based, a key factor is the level of abstraction offered to the user. This term covers a number of factors, including the ease of expressing the (essentially mathematical) problem to be solved, and the ease of expressing the (parallel) algorithms necessary to solve it. We use NWChem [14, 16] as an example to highlight the importance of the model's level of abstraction in enhancing productivity in developing complex high-performance software. NWChem is a large (over a million lines of code) computational chemistry package that provides high-performance, scalable implementations of a broad spectrum of methods in computational chemistry. Development of NWChem began in 1993, at a time when the chemistry community had experimented with parallel computing, but had produced few general, scalable, high-performance parallel algorithms. Experience had shown that many quantum chemical methods could not be implemented easily in the traditional message-passing programming model. In addition, effective abstractions and parallel I/O techniques were needed for out-of-core chemistry algorithms. These challenges led to emergence of novel parallel programming tools that enabled rapid development and implementation of scalable algorithms in this science domain, namely the Global Array (GA) toolkit [1, 26], Disk Resident Arrays (DRA) [21], and Shared Files [22, 13]. When coupled with algorithms that appropriately consider the non-uniform memory access (NUMA) nature of modern high-performance computers, the GA model augmented with DRA has proven both very high performance and very expressive for algorithms of the type that appear in quantum chemistry. Indeed, at present, essentially all scalable parallel quantum chemistry packages utilize Global Arrays or an equivalent programming model rather than the two-sided message-passing programming model that dominates most other scientific domains. Though no quantitative data is available, the qualitative experience of the NWChem effort (now ten years old, and embodying far in excess of 100 person-years of effort) is that the high-level abstractions provided by GA, DRA, and Shared Files were found invaluable in rapid development of scalable algorithms for this scientific domain, and quickly enabled scientists without prior experience in parallel programming to become productive contributors in this large software development effort. As previously noted, despite the continued popularity of the message-passing model in other fields, all scalable quantum chemistry codes use GA-like programming models.

In addition to level of abstraction and scalability of a programming model, a third, related, dimension is the generality of the programming model – whether it is appropriate to a narrow or broad range of computational problems and scientific domains.



**Figure 1.** Relative classification of programming methodologies with respect to level of abstraction, generality, and parallel scalability

While it is hard or impossible to precisely quantify abstraction level, generality, and scalability of various programming models without reference to a particular class of problems and other factors, it is possible to estimate rough relative positions of various programming models within this three dimensional space. By way of example, Figure 1 presents such an assessment:

- MPI [3] provides a very general, but rather low-level programming model and generally supports a high degree of optimization and tuning, making it possible to obtain performance close to the raw capabilities of the underlying hardware. It thus scores high with respect to model-generality and scalability, but ranks low regarding the abstraction-level offered to the software developer.
- The Global Arrays [1, 26] library-based approach offers a global shared view of multi-dimensional array objects that can be accessed by processes via block get/put/update operations. It inter-operates with MPI and provides comparable performance and scalability. Through its shared global view of array objects, it offers a higher level of abstraction to the programmer. However, since it only applies to array objects, the GA model is less general than MPI.
- OpenMP [4] is a completely general parallel programming model that offers a shared-space view of arbitrary data structures. Thus it ranks very highly along the dimension of generality, on par with MPI. We rank it

slightly higher than MPI and GA with respect to abstraction-level (referring in this case just to the ease of expressing the parallelism of the problem) However, scalable implementations of OpenMP for large-scale systems are yet to be realized.

- Automatic parallelization of standard sequential C/Fortran programs: Standard sequential programming languages like C, C++ and Fortran have been heavily used for developing scientific and engineering applications. There has been a long history of efforts to automatically parallelize sequential programs. Although there was great optimism in the early days of parallel computing that compiler techniques could be developed to automatically parallelize sequential programs, today the prospects of achieving such a goal seem very dim. Several vendors have marketed commercial auto-parallelizing compilers, but their effectiveness has been limited, especially in the context of highly parallel systems.

- PETSc (Portable Extensible Toolkit for Scientific computation [5]) is an example of a class of tools that facilitates the parallel (as well as serial), numerical solution of PDEs that require solving large-scale, sparse nonlinear systems of equation. The user creates and manipulates matrix objects, whose underlying representation and distribution among nodes of a parallel machine are transparent to the user. A variety of linear and non-linear solvers are implemented. The level of abstraction is very high, since both the data distribution as well as the parallel nature of the underlying solvers can be completely transparent to the user. It ranks rather low with respect to generality, since the high level of abstraction is only available for the set of numerical methods implemented.

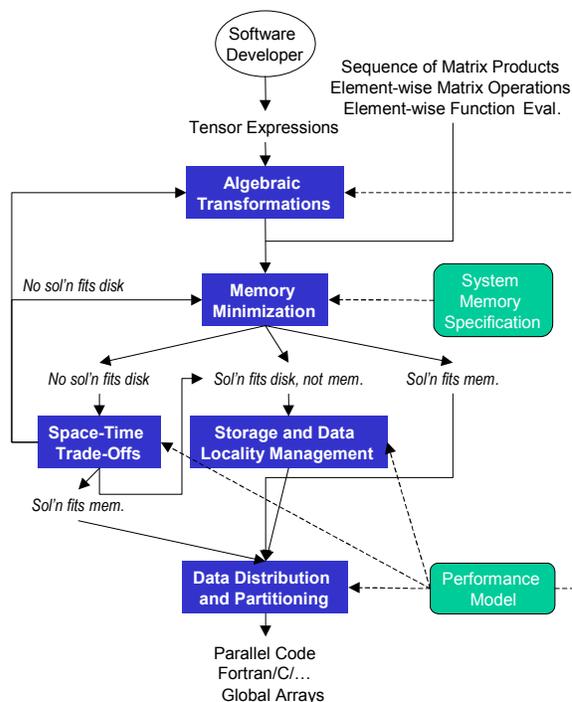
From the viewpoint of the scientist/software developer, one might describe the “holy grail” of productive scientific computing as being able to write the equations for the problem to be solved in a form that is close or identical to the way they would be expressed in a scientific paper and have tools turn this input into efficient, high-performance code. From the computer science viewpoint, the “holy grail” would be a programming model that maximizes all three axes (high abstraction/high generality/high performance), this is a daunting challenge (even for sequential computing!). However in order to address the looming crisis of software complexity, it is imperative to make progress toward solving this problem.

Such efforts typically try to move along one or more of the three dimensions, while maintaining the level of the remainder. In the remainder of this paper, we present examples of two efforts, taking different paths in the effort to raise the level of abstraction while preserving scalability. One involves the development of a high-level

language and optimizing compiler called the Tensor Contraction Engine (TCE) for a class of problems in computational chemistry, and the other an effort to generalize the Global Array programming model to transparently manage multiple layers of memory hierarchy.

### 3. The Tensor Contraction Engine

The Tensor Contraction Engine (TCE) is a domain-specific program synthesis system [6] being developed by a team of computer scientists and computational chemists. It is a system to automatically transform a high-level description of a quantum chemical model expressed in terms of complex tensor contraction expressions (essentially generalized matrix products on multidimensional arrays) into optimized parallel programs. A primary reason for the development of the system was to significantly decrease the amount of time needed to develop high-performance codes implementing accurate models for correlated electronic structure methods in computational chemistry packages. In this case, the level of abstraction (particularly the ease of expressing the problem itself) is so high that writing a program in the TCE environment is little more than writing out the tensor contraction expressions that define the method to be implemented and the parallelism is implicit in that input, but of course the TCE is limited to a



**Figure 2.** A schematic representation of the Tensor Contraction Engine’s architecture

narrow class of problems, as shown in Figure 1. Figure 2 provides a high-level picture of the transformation system. A brief description of the components of the system follows.

### 3.1. High-level language

The input to the synthesis system is a sequence of tensor contraction expressions (essentially sum-of-products array expressions) together with declarations of index ranges and symmetry and sparsity of matrices. The high-level notation offers two significant advantages:

1. For the user, the high-level representation makes it extremely convenient to express complex tensor contraction expressions.
2. For the compiler, the high-level representation provides essential information that facilitates domain-specific optimizations; such information would be difficult or impossible to extract out of code implementing such expressions in a language such as Fortran or C.

Figure 3 shows an example of a TCE program representing a term in a tensor contraction expression. It is shown along with a larger expression from a coupled-cluster [19, 20] model, shown in a notation used by quantum chemists in describing the computation. The tensor contraction expressions for accurate electronic structure models can have hundreds of such terms, and the Fortran codes implementing them often have tens of thousands of lines of code.

### 3.2. Algebraic transformations

Input from the user in the form of tensor expressions is transformed into a computation sequence. The properties of commutativity and associativity of addition and multiplication and the distributivity of multiplication over addition are used to search for various possible ways of applying these properties to an input sum-of-products expression. A combination that results in an equivalent form of the computation with minimal operation cost is generated. The problem of determining an equivalent operation-minimal form of the expression is NP-complete, but efficient pruning-search procedures have been developed that are very effective in practice [18].

### 3.3. Memory minimization

The operation-minimal computation sequence synthesized by applying algebraic transformation might require an excessive amount of memory due to the need to use large temporary intermediate arrays. The Memory Minimization step seeks to perform loop fusion transformations to reduce the memory requirements.

```

range V = 3000;
range O = 100;

index a,b,c,d,e,f : V;
index i,j,k : O;

mllimit = 1000000000000;

function F1(V,V,V,O);
function F2(V,V,V,O);

procedure P(in T1[O,O,V,V], in T2[O,O,V,V], out X)=
begin
  X == sum[ sum[F1(a,b,e,k) * F2(c,f,b,k)
    * sum[T1[i,j,c,e] * T2[i,j,a,f], {i,j}],
    {a,e,c,f}];
end

```

$$\begin{aligned}
 A3A = & \frac{1}{2} (X_{ce,af} Y_{ae,cf} + X_{ce,af} Y_{ae,cf} + X_{ce,af} Y_{ae,cf} + X_{ce,af} Y_{ae,cf} \\
 & + X_{ce,af} Y_{ae,cf} + X_{ce,af} Y_{ae,cf} + X_{ce,af} Y_{ae,cf} + X_{ce,af} Y_{ae,cf}) \\
 X_{ce,af} = & \langle ij | t_{ij}^{ce,af} \rangle \quad Y_{ae,cf} = \langle ab || ek \rangle \langle cb || fk \rangle
 \end{aligned}$$

**Figure 3.** An example of the typical representation of tensor contraction expressions used in the scientific literature (inset) together with sample of the input language for the Tensor Contraction Engine for one term in the inset expression

Optimal loop fusion is also an NP-complete problem [12]. An abstraction called the fusion-graph has been developed and has served as the basis for a search process used to evaluate alternate the loop fusion choices in the context of the TCE [17]. The loop fusion transformations along with array contractions to minimize memory are done without incurring any increase on the number of arithmetic operations.

### 3.4. Space-time transformation

If the memory minimization step is unable to reduce memory requirements of the computation sequence below the available disk capacity on the system a space-time trade-off is performed. This is done by exploring different ways of adding redundant loops that enable additional fusion and array contraction. The redundant loops increase the amount of computation, but additional array contractions so enabled might reduce space requirements of intermediate temporaries. Loop tiling can be used with the redundant loops to allow additional space-time trade-off. The fusion graph framework has been used to develop a search procedure to seek the best choice of redundant loops and tile sizes that can fit the computation within the available storage while incurring a minimal computational overhead due to the redundant loops introduced [9].

### 3.5. Storage and data locality optimization

If the space requirement exceeds physical memory capacity, portions of the arrays must be moved between disk and main memory as needed, in a way that maximizes reuse of elements in memory. The same

considerations are involved in minimizing cache misses — blocks of data are moved between physical memory and the space available in the cache. Loop blocking is used to minimize disk-to-memory transfer overhead. The issue of tile-size optimization is discussed in [11].

### 3.6. Data distribution and partitioning

This component determines how best to partition the arrays among the processors of a parallel system. We assume a data-parallel model, where each tensor contraction is distributed across the parallel machine. The arrays are to be disjointly partitioned between the physical memories of the processors. The data distribution pattern that minimizes the total inter-processor communication in executing a sequence of tensor contractions is determined. The data partitioning issue is discussed in [10].

### 3.7. Code generation

The back end of the synthesis system provides the output as pseudo-code, Fortran or C code. The generated code can be either serial or parallel, using Global Arrays (GA). Though targeting a traditional message-passing or other programming model is also quite feasible, we have found that the abstraction of globally-addressable shared data provided by the GA programming model greatly simplifies conceptual and code generation issues involved in the interface between the TCE-generated code and the supporting infrastructure provided by existing quantum chemistry packages, such as NWChem. Depending on the circumstances, the synthesized code could also call highly-tuned, machine-specific Basic Linear Algebra Subprograms (BLAS) libraries, or optimized low-level functions from the existing quantum chemistry packages.

The TCE approach has already demonstrated tremendous productivity gains. Using the prototype version of the TCE, which does yet incorporate several optimizations, more than 20 different quantum chemical methods have been implemented in just a few weeks, many receiving their first-ever parallel implementation in this way [15]. At a very conservative estimate of three months of effort each it would have required more than five years of effort to implement all these methods by hand, representing a productivity increase on the order of 50-100 fold, not including the improvements in time to solution due to the availability of parallel implementations. The ratio size of the synthesized Fortran code to the input tensor contraction expressions (measured as number of characters of source code, excluding comments) is also typically around two orders of magnitude. Work is underway on the fully optimizing version of the TCE to implement optimizations targeted at enhancing the

performance and scalability of the synthesized parallel Fortran code.

## 4. Extended Global Arrays

A logical step to raise the level of abstraction of the GA+DRA model is to integrate the management of three layers of memory hierarchy -- distributed main memory, shared memory on the SMP node of a cluster, and secondary storage -- under a single programming interface in an environment which automatically manages the hierarchy through extensions to the compilers for traditional programming languages. This effort, which we call "Extended Global Arrays" (XGA), is currently in the design stage.

### 4.1. Global Arrays

The Global Arrays toolkit presents to the application developer a distributed data structure as a single object and allows access as if it resided in shared memory. These features help the developer raise the level of composition and increase code reuse. A higher level of composition reduces the amount of code that must be written and enables scientists to program in terms of physically meaningful concepts rather than low-level manipulation of distributed data and explicit communication. Thus, it makes scientists more productive and permits more time to be spent optimizing performance-critical algorithms and application kernels. GA programming model includes as a subset message passing; in particular, the programmer can use full MPI functionality on both GA and non-GA data. The library can be used in C, C++, Fortran 77, Fortran 90 and Python programs.

GA implements a shared-memory programming model in which data locality is managed by the programmer through explicit calls to functions that transfer data between a global address space (a distributed array) and local storage. In this respect, the GA model has similarities to distributed shared-memory (DSM) models that provide an explicit acquire/release protocol. However, GA acknowledges that remote data is slower to access than is local data and therefore allows data locality to be explicitly specified and hence managed. Another advantage is that GA, by optimizing and moving only the data requested by the user, avoids issues such as false sharing or redundant data transfers present in some DSM solutions. The GA model exposes to the programmer the hierarchical memory of modern high-performance computer systems, and by recognizing the communication overhead for remote data transfer, it promotes data reuse and locality of reference.

The GA toolkit provides extensive support for controlling array distribution and accessing locality information. Both task-parallel and data-parallel programming styles are possible. Task parallelism is supported through the one-sided (non-collective) 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. Atomic operations are provided that can be used to implement synchronization and ensure correctness of updates of overlapping array sections. 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 interfaces to the parallel linear algebra libraries such as Scalapack as well as the TAO optimization toolkit [7].

## 4.2. Disk Resident Arrays

The disk resident arrays (DRA) model extends the GA model to another level in the storage hierarchy, namely, secondary storage [NF1996]. It introduces the concept of a disk resident array - a disk-based representation of an array. It provides functions for transferring blocks of data between global arrays and disk arrays. Hence, it allows programmers to access data located on disk via a simple interface expressed in terms of arrays rather than files. The benefits of global arrays (in particular, the absence of complex index calculations and the use of optimized array communication) can be extended to programs that operate on arrays that are too large to fit into memory. By providing distinct interfaces for accessing objects located in main memory (local and remote) and on the disk, GA and DRA render visible the different levels of the memory hierarchy in which objects are stored. Hence, programs can take advantage of the performance characteristics associated with access to these levels.

## 4.3. SMP Arrays

So-called SMP Arrays (SA) can be used as a shared memory cache for latency sensitive distributed arrays in cluster environments based on collection of Symmetric Multiprocessor (SMP) nodes. Due to its cost effectiveness, SMP systems are used as building blocks for both commodity clusters as well as custom architectures (e.g., IBM SP, SGI Altix, NEC SX, Cray X1). SA arrays resemble global arrays except their scope is limited to an SMP node rather than entire parallel job running on a cluster. SA are related to the mirrored arrays, that were initially introduced as an extension to Global Array model in context of wide-area-network grid computing environments [23, 24, 25] and recently proposed for reducing communication overhead on

clusters [27]. In the latter context, shared memory mirroring is used to cache entire global arrays on every SMP node. The arrays are replicated across cluster nodes and distributed within each node. The goal is to take performance advantage of the shared memory, which constitutes the fastest interprocessor communication protocol, and use it as replacement for more expensive network communication. In the mirrored approach, the user is responsible for managing consistency of the cached data and collective operations on arrays are globally synchronized. The SA arrays do not involve global synchronization in collective operations and are created and managed independently on each SMP node.

## 4.4. Integrated Programming Framework

The evolution of programming models is driven by the fundamental tradeoffs between high productivity and performance requirements in context of evolving scalable architectures. On one hand, high productivity demands high-level of abstractions that insulate the programmer from specificity of the underlying hardware details and allow describe the underlying mathematical model in terms of collection of algorithms and appropriate data structures. However, achieving high performance and scalability is difficult if the essential characteristics of the hardware, in particular the memory hierarchy, are ignored.

Intelligent and automated management of data movement is a fundamental and unifying theme for the Extended Global Array interface we are developing. The goal is to have a single interface for managing data and high level representation of the mathematical algorithms operating on multidimensional arrays while the details on the underlying data movement between secondary storage, distributed memory, shared memory, and local memory are handled by the XGA implementation. XGA attempts to address this problem while relying on three elements:

- Compiler analysis and code transformation
- Performance model for GA, SA, DRA operations
- Information on resource availability and configuration (disk space, memory, processor affinity).

The basic idea is to translate XGA programs into SA/GA/DRA code while orchestrating data movement, caching, and redistribution so that the performance is maximized while satisfying the constraints on the available resources. XGA would allow from a single source to generate in-core and out-of-core codes while reducing the programmer effort and maintenance costs.

## 5. Discussion

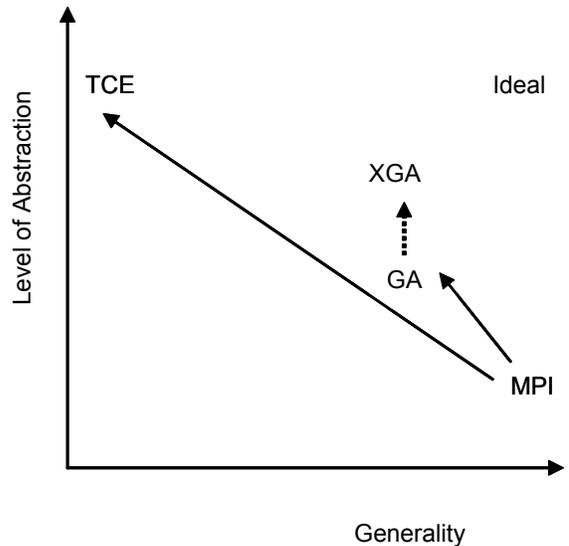
In this paper, we have discussed two efforts that we are currently engaged in, that seek to raise the level of abstraction offered to the developers of high-end software. Although message-passing with MPI can be used to develop and tune parallel programs in any application domain, we believe that the effort required to develop, validate and maintain very complex high-performance software is a deterrent and an impediment.

Historically, in the quantum chemistry domain, the need for higher-level abstractions to aid in coping with the complexity led to the development of the GA and DRA libraries; these libraries provide a programming model that has found many uses outside of the quantum chemistry domain as well. In the newer efforts described above, we are investigating other approaches to raising the level of abstraction while maintaining scalability and performance. The TCE is, once again, motivated by the needs of the quantum chemistry community, though in this case, the result is applicable to a relatively narrow domain because of the extremely high level of abstraction provided by the high-level language used. However, we believe that many of the approaches developed for the TCE can also be applied in the context of other more broadly applicable efforts at raising the level of abstraction in programming models for high-end computing. An example is the automatic memory hierarchy management aspect of XGA.

Figure 4 shows the relationships between these models in the two-dimensional abstraction/generalizability space. The third dimension of scalability can be made implicit when only considering models that achieve satisfactory levels of performance/scalability. The TCE uses GA in its implementation, but is not an *extension* of the GA model. The XGA effort, on the other hand is specifically an effort to extend the GA model to higher levels of abstraction. The two very different approaches to raising the level of abstraction of the programming model we have presented here have clear benefits to software productivity (some already realized, in the case of the TCE, and more expected following further development, in the case of both TCE and XGA). They also demonstrate the transferability of ideas in this space (automatic memory hierarchy management from TCE moving into XGA). Although the “holy grail” of a programming model with high level of abstraction, high generality and high scalability may be a distant goal, broad exploration of this space is likely to yield many new ideas with broad applicability and lead to the development of programming models that raise programmer productivity while delivering high performance.

In the future, we plan to explore ways of moving along the dimension of generality, while again maintaining scalability. Other approaches might seek to proceed along different paths in the three dimensional space of generality, abstraction-level and scalability, with the ultimate goal of developing very general-purpose programming language frameworks that offer high levels of abstraction and high scalability. However, there is a potential problem with approaches where the initial starting point has inadequate scalability, as exemplified by the experience with High-Performance Fortran [2]. A significant problem with HPF was that users were unable to achieve high performance for many applications with the initial releases of the compilers from vendors. This was because challenging compiler optimization problems had to be solved before performance could be delivered for a range of applications and this resulted in a vicious cycle. Vendors did not see it worthwhile investing in compiler optimization technology unless they perceived user demand; there was insufficient user demand without scalable performance.

The end goal of programming language models that rank high in all three dimensions is an extremely challenging one. Significant advances in compiler technology will be essential in achieving high scalability with general-purpose programming models offering high levels of abstraction. The sustained vision and support of governmental funding agencies towards this goal will be crucial. It will be very important for funding agencies to



**Figure 4:** GA and TCE are programming models at different levels of abstraction and generality, developed to make high-end software development easier than using MPI. XGA is a proposed model to further raise the level of abstraction above GA

engage in a long-term plan to support a variety of efforts that seek to advance the state-of-the-art in programming models offering high levels of abstraction, generality and performance. Progress will be greatly facilitated by sustained and strong interaction between application developers and systems software developers in vertically integrated teams, with expertise cutting across multiple layers: from the applications layer, programming models/frameworks layer, run-time layer, communications layer and hardware architecture.

## Acknowledgements

We wish to thank all of the members of the Tensor Contraction Engine collaboration and the Global Arrays team, without whose efforts this paper would not have been possible. This research is sponsored in part by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory, by the Office of Mathematical, Information, and Computational Sciences (MICS) of the U. S. Department of Energy Office of Science, Environmental Molecular Sciences Laboratory at PNNL, and by the National Science Foundation through the ITR program. ORNL is managed by UT-Battelle, LLC for the U. S. Department of Energy under Contract No. DE-AC05-00OR22725. PNNL is managed by the Battelle Memorial Institute for the U. S. Department of Energy under Contract No. DE-AC06-76RLO 1830.

## References

- [1] Global Array Toolkit Home Page, <http://www.emsl.pnl.gov/docs/global>.
- [2] High Performance Fortran Forum. High Performance Fortran Language Specification, Ver. 2.0, 1997. <http://dacnet.rice.edu/Depts/CRPC/HPFF/versions/hpf2/>.
- [3] Message Passing Interface Forum, "MPI: a message-passing interface standard," Intl. J. Supercomputer Appl. and High Perf. Comp. 8, 159-416 (1994).
- [4] OpenMP: Simple, Portable, Scalable SMP Programming, <http://www.openmp.org>.
- [5] Satish Balay, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Matt Knepply, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc Users Manual, Technical Report ANL-95/11 Revision 2.1.5, Argonne National Laboratory, 2002.
- [6] G. Baumgartner, D.E. Bernholdt, D. Cociorva, R. Harrison, S. Hirata, C. Lam, M. Nooijen, R. Pitzer, J. Ramanujam and P. Sadayappan. A High-Level Approach to Synthesis of High-Performance Codes for Quantum Chemistry. Proc Supercomputing 2002, 2002.
- [7] Steve Benson, Lois Curfman McInnes, Jorge J. More, and Jason Sarich. TAO Users Manual, Technical Report ANL/MCS-TM-242-Revision 1.5, Argonne National Laboratory, 2003.
- [8] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. Parallel Programming in OpenMP. Morgan Kaufman, 2000, ISBN 1-55860-671-8.
- [9] D. Cociorva, G. Baumgartner, C. Lam, P. Sadayappan, J. Ramanujam, M. Nooijen, D. Bernholdt, and R. Harrison. Space-Time Trade-Off Optimization for a Class of Electronic Structure Calculations. Proc. of ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI), June 2002, pp. 177-186.
- [10] D. Cociorva, X. Gao, S. Krishnan, G. Baumgartner, C. Lam, P. Sadayappan, J. Ramanujam. Global Communication Optimization for Tensor Contraction Expressions under Memory Constraints. Proc. of 17th International Parallel & Distributed Processing Symposium (IPDPS), Apr. 2003.
- [11] D. Cociorva, J. Wilkins, G. Baumgartner, P. Sadayappan, J. Ramanujam, M. Nooijen, D.E. Bernholdt, and R. Harrison. Towards Automatic Synthesis of High-Performance Codes for Electronic Structure Calculations: Data Locality Optimization. Proc. of the Intl. Conf. on High Performance Computing, Dec. 2001, Lecture Notes in Computer Science, Vol. 2228, pp. 237-248, Springer-Verlag, 2001.
- [12] Alain Darté. On the complexity of loop fusion. In Parallel Computing, Vol. 26, No. 9, 1175-1193.
- [13] Holger Dachsel, Jarek Nieplocha, and Robert Harrison. An Out-of-Core Implementation of the COLUMBUS Massively-Parallel Multireference Configuration Interaction Program. Proceedings of the 1998 ACM/IEEE Conference on Supercomputing, San Jose, CA, pp. 1-10, 1998.
- [14] High Performance Computational Chemistry Group, NWChem, A Computational Chemistry Package for Parallel Computers, Version 4.5 (2003), Pacific Northwest National Laboratory, Richland, WA 99352, <http://www.emsl.pnl.gov/docs/nwchem>
- [15] S. Hirata, Tensor contraction engine: abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories., The Journal of Physical Chemistry A, ASAP Article 10.1021/jp034596z S1089-5639(03)04596-1.2.
- [16] Ricky A. Kendall, Edo Apra, David E. Bernholdt, Eric J. Bylaska, Michel Dupuis, George I. Fann, Robert J. Harrison, Jialin Ju, Jeffrey A. Nichols, Jarek Nieplocha, T. P. Straatsma, Theresa L. Windus, and Adrian T. Wong, "High Performance Computational Chemistry: Overview of NWChem, a Distributed Parallel Application," Comp. Phys. Comm. 128, 260-283 (2000).
- [17] C. Lam, D. Cociorva, G. Baumgartner and P. Sadayappan. Optimization of Memory Usage and Communication Requirements for a Class of Loops Implementing Multi-Dimensional Integrals. Proc. 12th LCPC Workshop San Diego, CA, Aug. 1999.

- [18] C. Lam, P. Sadayappan and R. Wenger. On Optimizing a Class of Multi-Dimensional Loops with Reductions for Parallel Execution. *Par. Proc. Lett.*, (7) 2, pp. 157-168, 1997.
- [19] T. J. Lee and G. E. Scuseria. Achieving chemical accuracy with coupled cluster theory. In S. R. Langhoff (Ed.), *Quantum Mechanical Electronic Structure Calculations with Chemical Accuracy*, pp. 47-109, Kluwer Academic, 1997.
- [20] J. M. L. Martin. In P. v. R. Schleyer, P. R. Schreiner, N. L. Allinger, T. Clark, J. Gasteiger, P. Kollman, H. F. Schaefer III (Eds.), *Encyclopedia of Computational Chemistry*, Wiley & Sons, Berne (Switzerland). Vol. 1, pp. 115-128, 1998.
- [21] J. Nieplocha and I. Foster. Disk Resident Arrays: An Array-Oriented I/O Library for Out-of-Core Computations. *Proc. 6th Symposium on the Frontiers of Massively Parallel Computing*, Anapolis, MD, March 1996.
- [22] Jarek Nieplocha, Ian Foster, and Rick A. Kendall. ChemIO: High-Performance Parallel I/O for Computational Chemistry Applications, *Intl. J. Supercomp. Apps. High Perf. Comp.* 12, 345-363 (1998).
- [23] J. Nieplocha and R. J. Harrison. Shared memory NUMA programming on I-WAY. *5th International Symposium on High Performance Distributed Computer (HPDC-5)*, pp. 432-441, 1996.
- [24] J. Nieplocha and R. J. Harrison. Shared-Memory Programming in Metacomputing Environments: The Global Array Approach. *J. Supercomputing*, 11, 119-136 (1997).
- [25] J. Nieplocha, R. J. Harrison, and I. Foster. Explicit Management of Memory Hierarchy. In L. Grandinetti, J. Kowalik, and M. Vajtersic (Eds.), *Advances in High Performance Computing*, Kluwer Academic, NATO ASI Series #30, pp. 185-198, 1996.
- [26] J. Nieplocha, R.J. Harrison, and R.J. Littlefield, "Global Arrays: A Non-Uniform-Memory-Access Programming Model for High-Performance Computers," *J. Supercomp.* 10, 169 (1996).
- [27] B. Palmer, J. Nieplocha, and E. Apra. Shared Memory Mirroring for Reducing Communication Overhead on Commodity Networks. *5th International Conference on Cluster Computing (CLUSTER 2003)*, Kowloon, Hong Kong, December 2003.