A Block-Oriented Language and Runtime System for Tensor Algebra with Very Large Arrays

Beverly A. Sanders*, Rod Bartlett†, Erik Deumens‡, Victor Lotrich‡, and Mark Ponton‡

*Department of Computer and Information Science and Engineering, University of Florida, sanders@cise.ufl.edu
†Department of Chemistry, University of Florida, {deumens,bartlett}@qtp.ufl.edu
‡ACES QC, Gainesville, FL. {lotrich,ponton}@qtp.ufl.edu

Abstract—Important classes of problems in computational chemistry, notably coupled cluster methods, consist of solutions to complicated expressions defined in terms of tensors. Tensors are represented by multidimensional arrays that are typically extremely large, thus requiring distribution or in some cases backing on disk. We describe a parallel programming environment, the Super Instruction Architecture (SIA) comprising a domain specific programming language SIAL and its runtime system SIP that are specialized for this class of problems. A novel feature of the programming language is that SIAL programmers express algorithms in terms of operations on blocks rather than individual floating point numbers. Efficient implementations of the block operations as well as management of memory, communication, and I/O are provided by the runtime system. The system has been successfully used to develop ACES III, a software package for computational chemistry.

Keywords—parallel, very large arrays, domain specific language, computational chemistry, blocked arrays, tiled arrays, MPI, petascale

I. INTRODUCTION

A holy grail of parallel computing is to find ways to allow application programmers to express their algorithms at a convenient level of abstraction and obtain good performance when the program is executed. In addition, it is desirable that parallel applications can be easily ported to new architectures as new systems become available. In this paper we describe an approach to this problem for computations dominated by tensor algebra with very large dense arrays. The work was motivated by the concrete need for an effective environment to implement parallel software for coupled cluster methods for electronic structure prediction, an important class of problems in computational chemistry.

Most problems of interest in this domain will contain arrays that are too large to fit into the memory of a single processor, and sometimes are too large to be completely contained in the collective memory of all the processors, requiring them to be stored on disk. Thus, these arrays will necessarily be broken up into blocks and distributed throughout the system. Effective management of the memory at individual nodes, as well as management of the distributed arrays is crucial.

The algorithms used in this domain are extremely complex and in many cases more significant improvements in performance can be obtained by algorithmic improvements than tuning the details of the parallel program. Thus it is important that a programming environment offer a convenient level of abstraction for expressing algorithms that makes it easy for computational chemists to experiment with different algorithms.

The Super Instruction Architecture (SIA) is an attempt to provide such an environment. It comprises a domain specific programming language, Super Instruction Assembly Language (SIAL, pronounced "sail") and its runtime system, the Super Instruction Processor (SIP). Computational chemists express their algorithms in SIAL, a simple parallel programming language with a parallel loop construct and intrinsic support for distributed and disk backed arrays. The SIP provides efficient implementations of computational kernels, management of memory, communication, distributed and disk-backed arrays, and I/O.

An important feature of SIAL is that algorithms are expressed in terms of blocks (or tiles) of multidimensional arrays rather than individual floating point numbers. Although blocking arrays is a well-known technique in parallel programming, it is rarely supported at the programming language level. Expressing algorithms in terms of blocks is very natural in this domain and has two significant consequences for performance: data is handled at a granularity that can be efficiently moved around, and computation steps will be time consuming enough for the runtime system to be able to effectively and automatically overlap communication and computation. This also enhances programmer productivity by eliminating the need for tedious and error prone index arithmetic. The block size is an important tuning parameter and is not visible in SIAL, but rather chosen by the runtime system or a knowledgeable user as a runtime parameter, enhancing the portability of SIAL programs.

SIAL programs are compiled into SIA bytecode, which is interpreted by the SIP. The SIP is a parallel virtual machine that manages the complexities of dealing with parallel hardware, including communication and I/O, and also provides efficient implementations of the super instructions, which are computationally intensive operations on blocks. Super instructions simply take blocks as input and generate new blocks as output and do not involve communication. They are implemented in Fortran or another general purpose programming language and thus can take advantage of high quality optimizing compilers. Application development involves determining what the super instructions should be and implementing them if they are not already available. Then
SIAL is used as a high level scripting language to orchestrate the computation.

When a new architecture becomes available, the SIP needs to be ported and tuned for the new architecture while the SIAL programs themselves remain unchanged. The SIA has been implemented and ported to several different architectures and has been successfully used to implement ACES III [1]–[3], a software package for computational chemistry which provides parallel implementations of several advanced methods for electronic structure computations. ACES III is not a toy application or academic exercise; it is a serious tool used by computational chemists that pushes boundaries in both high performance computing and computational chemistry. It is available for download [1] under the GNU GPL.

The contributions of the work include

- An approach to manage the complexity of parallel applications by separating it into a high level scripting language, a runtime system that manages parallelism, load balancing, etc. and a set of computational kernels.
- Novel programming language constructs that support "programming with blocks".
- A runtime system that supports these language constructs in the context of very large arrays requiring efficient memory management at individual nodes.
- Excellent parallel performance, portability, and enhanced programmer productivity demonstrated by using the SIA to implement ACES III, a software package for computational chemistry.

II. CHARACTERISTICS OF THE PROBLEM DOMAIN

Computational chemists attempt to gain understanding of the nature of materials by computing properties of molecules via approximate solutions to Schrödinger’s equation. A variety of methods exist with different trade-offs between accuracy and computational requirements. Coupled cluster methods [4] are considered the "gold standard" method: they are the most computationally intensive but also the most accurate. The computational results are often used to augment, and sometimes predict, experimental results.

A typical calculation in this domain takes as input the geometry of a molecular system and a choice of single particle orbitals as a basis to expand the many-electron quantum-mechanical wave function. Complex algorithms produce properties of the molecular system such as total energy, electron density distribution, bond lengths and bond angles, rotational and vibrational spectra, electronic excitation energies, dipole moments, etc. Well-known approaches include the RHF and UHF (restricted and unrestricted Hartree-Fock), CCSD (coupled cluster with single and double excitations), CCSD(T) (coupled cluster with single and double excitation with triples computed by perturbation theory), and MP2 (second order Møller-Plesset perturbation theory) methods.

Generally the computational effort (floating point operations) increases as $n^5$ for MP2, $n^6$ for CCSD, or $n^7$ for CCSD(T), where $n$ is number of single-particle basis functions for the electrons. The amount of data that must be stored and accessed during the calculation depends on the algorithm. A minimum for CCSD is a few copies of the so-called $T$-amplitudes which is a four-index array of size $n^3N^2$, where $n$ is as before and $N$ is the number of electrons in the system. Typically $n = 10N$ in high-accuracy calculations. The number of copies ranges from a minimum of 2, to 10 when a convergence acceleration algorithm is used. In addition, arrays of size $n^2N^2$, and $n^3N$ are needed to store integrals used in the computation. For typical molecules of interest these values are around $n = 1000$ and $N = 100$. requiring around $n^2N^2 = 10^{10}$ double precision numbers or 80 GBytes per array for storage. Since about a dozen of these are needed that total requirement is about 1 TB. Of these arrays, three need rapid access and are usually distributed in RAM, the rest are used less frequently (once per iteration by the convergence acceleration) and are usually kept on disk. The larger array for integrals, used in some algorithms, requires 800 GB by itself.

III. SUPER NUMBERS (BLOCKS) AND SUPER INSTRUCTIONS

Numerous middleware systems and parallel programming languages provide some sort of abstraction of global shared memory for distributed arrays. While the details vary (see section VII), the abstraction helps to relieve the programmer from the tedious bookkeeping involved in locating and moving data in the system. Although mechanisms are available to move data in large chunks, and indeed this must be done in order to achieve acceptable performance, algorithms must still be expressed in terms of individual floating point numbers. This means that the programmer must first copy a chunk of the array to local memory, then write the algorithm in terms of accesses to local memory. In a few cases, the compiler performs automatic transformations to incorporate blocking, but these have not been successfully used for extremely large arrays.

The SIA departs from this approach by introducing the notions of super numbers (blocks, or tiles) and super instructions. Each dimension of a large array is broken up into segments which in turn define blocks of floating point numbers which can be thought of as super numbers. Super numbers are operated on by super instructions which take one or two super numbers, or blocks, as input, and generate a new super number. (For brevity, we will usually use the term block instead of super number.) Because the arguments are blocks, each super instruction performs a substantial amount of computation. In SIAL programs, loops iterate over segment numbers rather than the indices of individual elements. Algorithms are written in terms of super numbers and super instructions that operate on blocks of data thus ensuring that the algorithm is expressed in a way that can be effectively parallelized. SIAL source programs are independent of implementation specific details such as the number of processors, the location of data, and the

1Notable exception include Hierarchical Tiled Arrays [5] and FLAME [6], [7].

2It is possible to access individual numbers in SIAL programs, but this should be rare as it impacts performance negatively.
segment size. Super instructions are implemented efficiently in the runtime system.

As an example, consider the following term expressing the contraction\(^3\) of two four dimensional tensors (indexed by \(\mu, \nu, \lambda, \) and \(\sigma\); and by \(\lambda, \sigma, i, \) and \(j, \) respectively) and yielding a third four dimensional tensor (indexed by \(\mu, \nu, i, \) and \(j\)).

\[
R_{ij}^{\mu\nu} = \sum_{\lambda\sigma} V_{\lambda\sigma}^{\mu\nu} T_{ij}^{\lambda\sigma}
\]  

(1)

This could be expressed as a contraction on blocks as

\[
R(M, N, I, J)^{\mu\nu} = \sum_{LS} \sum_{\lambda \in L} \sum_{\sigma \in S} V(M, N, L, S)^{\mu\nu} T(L, S, I, J)^{\lambda\sigma}
\]

(2)

The indices \(M, N, L, S, I, J\) are block indices; the block \(V(M, N, L, S)\) is itself a 4-index matrix containing \(seg^4\) elements where \(seg\) is the segment size, which typically would be chosen to be between 10 and 50. The contraction of one block of \(V\) with one block of \(T\), which produces one block of \(R\), is one super instruction which requires \(2 \times 10^6\) to \(2 \times 2,500^4\) floating point operations. The SIAL source program does not access the indices inside the block; these are only visible inside the super instruction.

Because blocks are the units of data that are moved around the system between processors, by choosing an appropriate block size, the application can be tuned to make most effective use of the system’s memory capacity, caching architecture, and communication infrastructure.

In order to obtain good performance, it is crucial to use asynchronous communication to overlap communication and computation. In the SIA, overlapping computation and communication is handled at runtime by the SIP. The SIP recognizes the loops that provide opportunities for effective overlapping and prefetches blocks that will soon be needed. In a well-tuned SIAL program, a large portion of the communication is hidden behind computation. Segment sizes are either a default value that has been chosen the particular system or that is given a list of index variables and an optional list of ranges that may be defined using a symbolic constant that is determined during program initialization. As a result, the size of an array is known and fixed during the program execution, but need not be known when the program is written.

There are three types of indices: segment indices\(^4\) count segments and enable programming in blocks; simple indices, most commonly used to count iterations; and subindices. A subindex is related to a specific segment index and allows access to subblocks. Subindices will be discussed in more detail in section IV-E.

Blocks of distributed arrays are obtained with \texttt{get} and stored with \texttt{put} commands. Blocks of served arrays are obtained with \texttt{request} and stored with \texttt{prepare} commands.

SIAL provides a set of intrinsic super instructions that make working with blocks convenient. When scalars are assigned to a variable representing a block, all elements of the block receive that value. Similarly, when a block is multiplied by a scalar, all elements of the block are multiplied by the scalar, etc. An assignment such as \(V1(K,J,I) = V2(I,J,K)\), permutes \(V2\) and assigns the result to \(V1\).

### B. Expressing parallelism and other control structures

Parallelism is explicitly expressed using a \texttt{pardo} command that is given a list of index variables and an optional list of \texttt{where} clauses, each with a boolean expression. The SIP executes iterations, in parallel, over all possible combinations of the values in the range of the given indices that also satisfy the \texttt{where} clauses. The \texttt{where} clause is most frequently used to eliminate redundant computations with symmetric arrays.

\texttt{Pardo} loops are not allowed to be nested syntactically, but because there are no implied barriers in SIAL, two \texttt{pardo} loops not separated by a barrier may be scheduled to execute

\(^3\)Tensor contraction operations occur frequently in the domain and are defined as follows: Let \(\alpha, \beta, \) and \(\gamma\) be mutually disjoint, possibly empty lists of indices of multidimensional arrays representing the tensors. Then the contraction of \(A[\alpha, \beta]\) with \(B[\beta, \gamma]\) yields \(C[\alpha, \gamma] = \sum_{\beta} A[\alpha, \beta] \ast B[\beta, \gamma]\).

\(^4\)There are actually several segment index "types" corresponding to domain specific concepts. For example, \texttt{aoindex} and \texttt{moindex} represent atomic orbital and molecular orbital. This allows the type system to perform useful checks on the consistent use of index variables.
in parallel. SIAL programmers have no control over how a *pardo* loop is executed; all scheduling is done by the SIP.

Other control structures include a *do* loop, which is given a single index variable and conducts a sequential iteration over the range of the index variable, if and *if else* commands, and procedure calls.

**C. Super Instructions**

A SIAL programmer has a rich collection of super instructions at his or her disposal. Some super instructions, such as the block contraction operator, are intrinsic. Non-intrinsic super instruction can be added to the SIP without changing the SIAL language itself and are invoked from SIAL programs using the *execute* command. When changing to a new class of algorithms, application programmers will typically develop additional super instructions.

In addition to computational super instructions, super instructions are provided for a variety of operations including I/O, barriers, and utility functions. Two types of barrier super instructions are provided, one that should be used between conflicting accesses to distributed arrays, and one that should be used between conflicting accesses to served arrays. The runtime system detects most improper uses of barriers.

The super instructions, *blocks_to_list list_to_blocks serialize* and *deserialize* distributed arrays. This facility is used to pass data between different SIAL programs. It is also used to provide a rudimentary checkpointing facility that allows programs to be restarted from a checkpoint if the program is unable to run to completion.

**D. Example**

The following SIAL fragment computes the term shown in equation 2 from Section III.

```sial
pardo M,N,I,J
tmpsum(M,N,I,J) = 0.0

do L
do S
    get T(L,S,I,J)
    compute_integrals V(M,N,L,S)
    tmp(M,N,I,J) =
        V(M,N,L,S) * T(L,S,I,J)
tmpsum(M,N,I,J) += tmp(M,N,I,J)
enddo S
enddo L

put R(M,N,I,J) = tmpsum(M,N,I,J)
endpardo M,N,I,J
```

The *pardo* *M,N,I,J* statement specifies parallel execution of the loop over the segment indices *M,N*I and *J*. Their declarations have been omitted; the ranges of the indices were given when the index variables were defined. For example, we might have a declarations such as *aoindex M=1, norb* where *norb* (number of orbitals) is a symbolic constant and

**E. Subindices**

As described earlier, SIAL handles large arrays by partitioning each dimension into segments. In SIAL source code, one might find a reference to, say, *X(i,j)*. Since *i* and *j* are segment indices, *X(i,j)* refers to a block of data. For example, if the segment size in both dimensions is 16, then *X(2,2)* actually refers to the block *X ([17:32],[17:32])* specified with normal indices. An operation like *X(i,j)+V(j,k)* is implemented by a super instruction that accesses the individual elements of the two blocks to perform the contraction. The segment sizes are chosen to give appropriate granularity in the application; data is moved in blocks, and operations on a block perform enough work that communication and computation can be overlapped. The segment size is fixed at initialization time and is not available in the SIAL source. This approach may fail in situations where arrays with too many dimensions are generated. For example, *A(a,b,c,k)+B(k,1,m,n)* generates a 6 dimensional result, say *C(a,b,c,1,m,n)*. The size of the six dimensional array is *seg*6, which, for typical segment sizes requires too much memory to be feasible. If the segment sizes are made smaller, then the blocks of A and B become too small for the rest of the computation to exhibit good performance.

6The SIA does not attempt to automatically fuse loops and in this case, which is typical, the time is dominated by the contraction and performance improvement from doing so would be negligible. If this were not the case, a new “fused” super instruction could be introduced.
The subindex construct allows subblocks of the blocks to be conveniently accessed.

1) Declaration of subindices: When declared, a subindex is not given a range, but is declared to be a subindex of a previously declared segment index. For example, in the declaration subindex ii of i, the superindex i must be a previously declared index. The number of subsegments per segment is determined by a runtime parameter in the same way as the segment size. If i has range \([low(i), high(i)]\) and segment size \(seg(i)\), then the range of ii is \([(low(i) - 1) * n + 1, high(i) * n]\) where \(n = seg(i)/seg(ii)\). The subindex inherits the type from its super index and may be used in declarations. An array declared with a subindex will contain the same number of data items as an otherwise equivalent array declared with its super index. For example, consider the following declarations.

\[
\begin{align*}
\text{moaindex } j & = 1, 4 \\
\text{moaindex } i & = 1, 4 \\
\text{subindex } ii & \text{ of } i
\end{align*}
\]

These indices could be used with the array shown in Figure 1. The following declarations define temporary blocks that would hold a complete block and subblock, respectively.

\[
\begin{align*}
\text{temp } & X(i, j) \\
\text{temp } & X(ii, j)
\end{align*}
\]

Fig. 1. Two dimensional array showing segment indices and subindices.

2) Slices and Insertions: Slices allow a subblock to be extracted from a block, while insertions do the opposite. With the above declarations, and assuming that i,j, and ii are appropriately defined at this point. The slicing assignment \(Xii(ii, j) = X(i, j)\) takes the indicated slice of the \(16 \times 16\) \(X(i, j)\) block and copies it into the \(4 \times 16\) \(Xii(ii, j)\) block. The insertion assignment \(X(ii, j) = X(ii, j)\) inserts the smaller subblock into the appropriate location in the larger block.

3) Subindices in loops: Typically, one acquires a particular block and then wants to iterate over the subblocks in that block. The do in and pardo in constructs allow this to be easily expressed. These loops iterate over the values of ii that fall within the block specified by i. The super index must have a well-defined value at the point where the do in or pardo in construct occurs, which implies that this construct must be nested inside a normal do or pardo loop over the super index. For example, consider the following code fragment.

\[
\begin{align*}
pardo & j \\
\text{do } i & \\
\text{do } ii & \text{ in } i \\
Xii(ii, j) & = X(ii, j) \\
\text{do something with } Xii(ii, j) & \\
\text{endo } ii & \\
\text{endo } i & \\
pardo & j
\end{align*}
\]

The pardo j loop will create four parallel tasks, one for each value in the declared range of j and corresponding to a row in Figure 1. The do i in line 2 loops over the blocks in a row, and do ii in i in line 3 loops over the subblocks within the i,j block. In practice, subindices are only needed in the occasional situations where arrays with dimensions greater than four are created during a computation.

V. SIP

The SIP (super instruction processor) is a parallel virtual processor and run-time environment written in C and Fortran with most implementations using MPI for communication. The focus of the SIP design effort was to produce a well-engineered system that can efficiently execute SIAL programs and be easily ported to and tuned for different systems. A design principle of the SIP is to maximize asynchrony—all message passing is asynchronous and all barriers are explicit.

A. SIAL byte code

A SIAL program is compiled into super instruction byte code which is executed by the SIP. The byte code includes a table of instructions to be executed along with operand addresses given as entries in data descriptor tables. A data descriptor for a distributed or served array contains the location of each block in the array. Some of the values in the tables are symbolic values that correspond to values of predefined constants. The symbolic values are replaced with a concrete value during initialization.

The instructions can be roughly classified into computational super instruction, control, I/O, and synchronization instructions. The compute super instructions include computationally intensive instructions; for example, a set of instructions to perform various tensor contractions between blocks. These instructions, which take two blocks as an argument and generate a third block as a result, should be implemented as efficiently as possible on the given platform. This might mean taking advantage of a powerful optimizing compiler for sequential code along with high quality implementations of library routines such as DGEMM, or using thread-level parallelism or co-processors.
While a useful set of computational super instructions have been provided, many with intrinsic language support, it is often necessary when implementing a new class of algorithms to create additional super instructions. This is easily done and does not require modifying the SIAL programming language.

Other instructions are control instructions for loops, if-else statements, etc. The instructions implementing the SIAL get, put, prepare and request statements may also initiate communication. The SIP first determines whether the indicated block is available at the current node. It may be available because it was assigned to be stored there, or because it is still available in the block cache from a recent use. If not, non-blocking communication is initiated to acquire or send the indicated block using information in the blocks data descriptor. As much as possible, instructions are executed asynchronously: Those involving communication are started and then control returns to the SIP task so that more computations or different communications can be performed. The SIP looks ahead and requests several blocks that it expects will be needed soon, thus overlapping communication and computation. When an instruction that needs a block executes, it will wait if the communication to acquire the block is still in progress.

B. Master, workers, and I/O servers

The SIP is organized as a master, a set of workers, and a set of I/O servers, each implemented (in most SIA implementations) using a sequential MPI process. When execution of a SIAL program is initiated, the master performs the management functions required to set up the calculation.

In order to gather information about the memory usage, the master inspects the SIAL program in "dry-run" mode. The dry run is an analysis which makes an estimate of the memory requirements for each worker given the number of processors that will be participating in the computation, the sizes of the arrays, and the distributed data layout. This feature allows the user to avoid wasting valuable supercomputing resources on an infeasible computation. Most data allocations depend on values determined at runtime, but the required calculations are relatively simple (compared to the hour-long electronic structure calculations) and are performed once for the dry run and then again later when needed. If the information from the dry run implies that the computation is not feasible with the available memory, this is reported to the user along with the number of processors that would be sufficient to perform the calculation.

The memory in each SIP worker is managed by dividing it into several stacks of preallocated blocks of memory of various sizes. For a segment size of $seg$, a $d$-dimensional block requires space for $seg^d$ floating point numbers. The number of blocks of each size is determined from information obtained during the dry run analysis. For good performance, the segment size should have the correct relation to the cache of the processor used.

In the current implementation, blocks of a distributed array are assigned to workers using a simple, static strategy. Because the applications tend to exhibit irregular access patterns with little spatial or temporal locality, and most of the communication should be overlapped with computation in any case, this works well in practice. If the algorithm is not overlapping communication with computation effectively, then solving that problem will make a much more significant difference in the performance than would more elaborate schemes to improve the data placement. Nevertheless, the approach to data distribution could be modified and improved at any time without requiring any change in the SIAL programs.

The master is responsible for allocating work to the workers. Recall that all parallelism is explicitly specified by the SIAL pardo instruction. Initially, the set of iterations, which is determined by the range of the indices and the where clauses, is divided into "chunks" and doled out to the workers. When a worker completes its chunk, it requests another chunk from the master. The chunk size decreases as the computation proceeds. This is similar to the approach taken with guided scheduling in OpenMP.

Each worker loops through the instruction table executing bytecode instructions, periodically checking for messages and processing them. Messages may be from the master and contain information about pardo chunks, from I/O servers containing requested blocks, or from other workers getting or putting blocks.

The I/O servers support the SIAL served arrays. Blocks are written to a served array with the prepare instruction. The worker determines the responsible I/O server and sends the block. Each I/O server contains a cache for served array blocks. Blocks arriving as a result of a prepare command are placed in the cache and lazily written to disk. Blocks that have been recently used or that have been prefetched may also be found in the cache and if requested by a worker can be immediately returned without waiting for disk I/O. Replacement is done using a LRU strategy.

All operations of an I/O server are non-blocking: all communication uses mpi_send and mpi_recv and all I/O uses asynchronous read and write. This prevents a single operation from blocking progress on an independent operation. Blocks are allocated in I/O server block pools or on a hard disk drive only when actually filled with data, simplifying the declarations of symmetric arrays.

An I/O server may also perform certain domain specific computations, namely computing blocks of integrals that provide a matrix representation of certain partial differential equations and are computed on demand rather than stored.

VI. EXPERIENCE

A. Portability

ACES III is available on several NSF TeraGrid, DOE, and DOD HPCMP MSRC sites with a variety of architectures including the SGI Altix SMP (pople.psc.teragrid.org), Cray XT3 (bigben.psc.teragrid.org), Cray XT4/XT5 (jaguarpf.ccs.ornl.gov), IBM Cluster 1600 with

\cite{8}
Power5+ processors (babbage.navo.hpc.mil), Linux Networx Advanced Technology Cluster (mjm.arl.hpc.mil), and Sun Opteron cluster (midnight.arsc.edu). A port to the BlueGene/P system at Argonne National Labs is in progress. Most recently, ACES III has been built and run on Power7 systems running both Linux and AIX.

Tuning the SIA for a new system involves two different tasks: tuning the SIP runtime, and tuning the super instructions implementing the computational kernels. The former has proved to be fairly easy when the systems are similar, say Linux with InfiniBand. The correct choice of segment size is the most significant factor. The BlueGene/P system is an exception: although obtaining a running system was not difficult and required only two days,\(^9\) tuning has required considerably more effort than previous ports due to significantly different processor/network performance ratios. It was necessary to modify the prefetching mechanism to avoid blocks arriving too early, causing eviction and refetching of blocks that would be reused. The performance improvement due after tuning was large. For example a test case that ran in 1,500 seconds on a Cray XT5 with 512 processors initially took more than 6 hours on the 512 cores of a BlueAGene/P. After tuning the SIP, the times are within a factor of four commensurate with the ratio of the processor speeds, indicating that the communication is no longer the rate determining factor.

The super instructions implementing the computational kernels can be tuned independently from each other and the rest of the SIP runtime as they do not involve communication and are implemented in Fortran or C. This work is currently being performed on the Power7 in preparation for porting ACES III to Blue Waters when it becomes available.

**B. Programmer productivity and tuning SIAL programs**

One of the design goals for the SIA was to be able to perform significant tuning for different platforms without the need for a major rewrite of application code every time. SIAL facilitates this by not exposing segment sizes to the SIAL program, thus this value can easily be adjusted for different platforms without changing the source code.

Although SIAL offers a level of abstraction that hides much of the complexity of a parallel computation from the computational chemists, it is still possible to write programs that perform poorly and that can be improved with better reuse of data, avoidance of redundant computation in loops, etc.

Each run of a SIAL program provides profile information that helps identify such situations. Recall that the SIA organizes the execution of the parallel program into steps, namely the execution of one super instruction, that take a non-negligible amount of time to complete. As a result it is possible to collect timing information for each super instruction without noticeable impact on application performance. Since the SIAL compiler itself does not perform any sophisticated optimization, the relationship between the source code and the profile data is transparent. Timing data collected includes execution time for pardo loops, procedures, and individual super instructions, and wait time for each pardo loop and the total wait time for the entire program. The wait time indicates how much time is spent waiting for blocks of data to become available. Small wait times indicate effective overlap of computation and communication. In most cases, the information collected will clearly point to what needs to be modified to improve performance.

Programs may also need to be retuned when used for significantly different problem sizes (i.e. bigger molecules, more basis functions). This type of tuning typically involves changing assumptions about the sizes of data structures made during the initial development of a SIAL program. For example, changing an array from distributed to served, introducing local arrays to compute temporary terms, etc.

Programs that, based on our experience, would have taken several months using a straight MPI implementation can be developed in a week or two by an experienced SIAL programmer, including performance tuning.

**C. Parallel Performance of ACES III**

Figure 2 shows scaling up to 256 processors for an RHF CCSD energy calculation for a luciferin \((C_{11}H_{8}O_{3}S_{2}N_{2})\) molecule run on a Sun cluster (midnight at ARSC), comprised of Opteron processors with an Infiniband interconnect. The top line is Scaling efficiency relative to the 32 processor computation, the middle line shows the average elapsed time for a CCSD iteration, and the bottom line shows the percentage of the elapsed time spent waiting for communication. Efficiency and the wait time as a percentage of the computation time are measured on the right vertical axis. In this example, which is typical, the average wait time ranged from 8.4 to 13.4 percent of the computation time. Specific pardo loops that take a large amount of time and thus contribute significantly to the overall execution time have been individually tuned in the SIAL program to achieve better than the average latency hiding. The differences for different numbers of processors are largely due to more or less fortuitous placement of data.

Figure 3 shows scaling up to 2048 and 4096 processors, respectively for an RHF CCSD energy calculation for a water cluster \((H_{2}O)_{21}H^+\) on a Cray XT5 (pingo at ARSC) and a Cray XT4 (kraken at NICS).

Figures 4, 5, and 6 give preliminary results from the DOE Cray XT5 Jaguar system at Oak Ridge National Labs. Figure 4 shows scaling and efficiency relative to the 1000

---

\(^9\)According to ALCF (Argonne Leadership Computing Facility) staff, it typically requires up to several weeks or months to get a newly ported parallel application to that point.
processor computation for RHF-CCSD for two molecules, RDX (C\textsubscript{3}H\textsubscript{6}N\textsubscript{6}O\textsubscript{6}) and HMX (C\textsubscript{4}H\textsubscript{8}N\textsubscript{8}O\textsubscript{8}). The larger HMX molecule displays much better strong scaling for CCSD. Figure 5 shows RHF-CCSD(T) for RDX with good strong scaling up to around 30,000 processors. Efficiency is relative to the 10,000 processor computation.

Figure 6 shows the scaling for the diamond nano crystal. We observe strong scaling up to 72,000 cores. Computations were performed with 84,000, 96,000, and 108,000 cores, but with longer execution times than with 72,000 cores. In the results shown in the figure, all runs were identical except for the number of processors. However, tuning by optimizing the segment size for each processor count yield better results. For example at 84,000 processors, the time drops to 57,534 seconds from 83,217. This is better than the 79,412 seconds at 72,000 cores indicating how easily ACES III can be tuned, and how important it is to have realistic problems with which to do the tuning.

Obtaining objective and reliable comparative benchmarks is even more difficult than scaling results. However, we do have some\textsuperscript{10} comparing the performance of ACES III and NWChem [9], a well-known software package for electronic structure calculations based on Global Arrays [10]. Figure 7 shows a UHF MP2 computation for Cytosine+OH (C\textsubscript{4}H\textsubscript{6}N\textsubscript{3}O\textsubscript{2}) computed on the SGI Altix 4700 (people at the Pittsburgh Supercomputing Center). The results show that ACES III with 1GB per core achieves better performance than NWChem with

\textsuperscript{10}NWChem results were reported by Shawn Brown at PSC. [8]
both 2GB and 4GB per core. More significantly, NWChem did not successfully complete the calculation within 24 hours on any of the tested number of processors with only 1GB per core, or with either 2GB or 4GB of memory with 16 processors. NWChem uses the Global Array Toolkit as the underlying data architecture which requires a very rigorous organization of the data blocks and communication patterns. The SIA has a much more adaptable data architecture that is programmed in a loose way in SIAL, giving the SIP more freedom to manage the memory and adapt to different systems.

The work on Hierarchical Tiled Arrays [5], like ours, supports programming with blocks. An HTA is a data type that allows recursive decomposition of arrays into blocks with support for dealing directly with the blocks (tiles). Dynamic partitioning and overlapped tiling are also supported. Although the reported implementation does support distribution of HTAs, much of the focus is on tiling arrays to obtain a better match with multilevel memory hierarchies within a node. In principle, the HTA concept should also work with disk-backed arrays, but this has not been implemented to our knowledge.

The FLAME project [6], [7] for dense linear algebra includes a notation for expressing algorithms, a methodology for systematic derivation of algorithms, Application Program Interfaces (APIs) for representing the algorithms in code, and various tools for mechanical derivation, implementation, and analysis of algorithms and implementations. The FLAME notation allows convenient “index-free” expression of algorithms for dense linear algebra. Both HTA and FLAME could be effectively used to implement SIA super instructions.

The Global Arrays (GA) toolkit [10], [16] developed for NWChem [9] and used in MOLCAS [17] and COLUMBUS [18] is a library that provides an abstraction of global, shared, multidimensional arrays. It was developed with a similar problem domain to SIAL in mind. Programmers use put and get routines to copy arbitrary rectangular sections of arrays between the shared array and local memory. GA implementations also support disk resident arrays for arrays too large to fit in the distributed memory of the system.

A well-tuned program for computational chemistry implemented in either a PGAS language or using GA would certainly lay out matrices in blocks, arrange for data to be moved around the system in blocks rather than remotely accessing individual array elements one at a time, and incorporate techniques to overlap communication with computation. However, the algorithms would still be written in terms of accesses to individual elements. The techniques used to achieve good performance must be incorporated manually and rely on the programmer to get all the details right. Using non-blocking communication requires explicit introduction of a split-phase barrier. In GA, it is natural to move data around in chunks, but the programmer must explicitly give the indices defining the chunks and must explicitly program overlapping communication with computation using non-blocking get (nga_nbget) and wait (nga_wait) functions. In the SIA, the idea that data is handled in blocks is intrinsic and efficient data movement and overlapping computation and communication happen as a matter of course.

An important feature of the PGAS languages and GA is that the data placement and work assignments are specified in the application programs. This control is generally considered an advantage as it allows the programmer to take advantage of locality to improve performance by reducing communication, but comes at the cost of building assumptions about the data locality in the programs. The Distributed Data Interface (DDI) [19] of GAMESS [20] is simpler in design than GA, but also is very specific in how the data is laid out. With the recent

Fig. 7. Cytosine+OH UHF MP2 gradient calculation by ACES III and NWChem performed on pople, the SGI Altix 4700 at the Pittsburgh Supercomputing Center.

**VII. RELATED WORK**

Several programming languages including Co-array Fortran [11], UPC [12], and Titanium [13] have gained much attention recently. These languages take the approach of extending a general purpose programming language to support the Partitioned Global Address Space model. In the PGAS model, a collection of threads operate in a single global address space that is logically partitioned between threads. Each thread has its own private space and the language provides convenient access to shared data, regardless of where it is located in the system. A key aspect of the model is the notion that each thread has affinity to a certain portion of the address space which is physically co-located with that thread, and thus can be accessed faster than data whose access requires communication. The language syntax provides a convenient and expressive way for the programmer to specify the data layout and allocation of work to threads, allowing locality to be exploited. UPC and Titanium support pointers to distributed data and can be used to implement applications utilizing a wide variety of data structures. We are not aware of a PGAS language that supports very large arrays that need to be resident on disk. The more recent general purpose PGAS languages, X10 [14] and Chapel [15], do not yet have implementations sufficiently mature to be used for production scientific code.
extension to optimize the use of shared memory nodes [21], some limitations have been alleviated, but still the programmer must make decisions about how to lay out the data structure in a very precise way. If the end user is then confronted with the situation where the program allocates data in a way that does not match the available computer system resources, the calculation will simply not run.

The SIA has fewer such limitations. The data array types available in SIAL are only specified to the extent that is needed to specify the algorithm correctly. The rest is left open. In SIA a significant part of the data placement decisions are left to the SIP and to the end-user of the application. This extra flexibility allows SIAL programs to run on more configurations than GA programs solving similar problems. An example was seen earlier in Figure 7.

SIAL is not the only attempt to introduce a domain specific language for electronic structure computations. The Tensor Contraction Engine (TCE) [22] provides a domain specific language that is closer to the mathematical notation used by computational chemists than SIAL. The TCE developers claim that it reduces the programming time from months to days or hours and about half of the code in the NWChem package [9] was generated using TCE. Briefly, TCE code generation occurs in three steps. The first translates a mathematical expression into a sequence of operations. The second analyzes the operations, removes redundancies by introducing suitable intermediate results, and optimizes loop structures. The final step generates Fortran code that uses the GA toolkit for communications and which may be manually tuned. A SIAL program is at approximately the same level of abstraction as the results of the second step of TCE. The transparency of the relationship between the SIAL source code and the resulting performance along with the simplicity of the SIAL syntax provide an advantage over generated Fortran/GA code when tuning at this level, especially when required changes are substantial.

We note that the SIA design is similar to that advocated by the Berkeley Parlab [23] which divides the task of developing parallel software into two levels which involve programmers with different kinds of expertise. The top level is the productivity layer used by domain experts which abstracts away from the details of the underlying parallel system. Below that is an efficiency layer that handles the details of the parallel implementation and is developed by parallel programming specialists. In the SIA, SIAL is the productivity layer and SIP is the implementation layer.

VIII. CONCLUSION

Experience implementing and using the SIA has led to the following observations.

- Language support for programming in blocks in the form of segment indices and subindices along with iterations over segments is an effective abstraction for problems in this domain.
- The SIAL language is not as general as Fortran and C/C++, but the restrictions allow a more efficient parallel implementation.
- Because basic operations are relatively time consuming, we can keep track of very detailed performance metrics without an impact on performance, or the need to use separate profiling tools.
- Because SIP has been designed to be fully asynchronous, it has been very easy to port to several different architectures as described in Section VI-A. The ports sometimes required switching from MPI with POSIX threads to shmem or to pure MPI. These changes were confined to small sections of code. The SIP also provides good performance on a NUMA; the SIP takes control of the data placement and movement from the OS, thus the SIP is using the switch network directly in ways similar to distributed memory architectures.
- The ease of writing algorithms in SIAL allows our SIAL developers to write multiple implementations of the same algorithm and use the two versions as tests of each other. Such development takes days, rather than weeks or months typical for development in Fortran or C/C++ with explicit MPI and POSIX threads calls.

The interface between SIAL and the SIP has turned out to be very effective. We have identified opportunities to enhance SIAL and also provide useful tool support for SIAL programmers. These include enhancing the expressiveness of the pardo loops and providing support for performance modeling. We have also identified areas where the SIP can be improved, especially to support scaling to petascale including improvements in the checkpointing facilities and more attention to data locality. The design of the SIA has clearly identified the computationally intensive kernels and encapsulated them in the super instructions. This prepares the SIA very nicely for exploitation of the new opportunities offered by general purpose graphical processor (GPGPU), Cell Broadband processors, and field programmable gate arrays (FPGA).

Finally, while our current experience is limited to the domain of computational chemistry, we believe that the main ideas underlying the SIA and, with some refactoring, its implementation can fruitfully be employed in other domains. Future effort will be directed towards this goal.

ACKNOWLEDGMENTS

Shawn Brown of the Pittsburgh Supercomputing Center ran the NWChem benchmarks reported in Figure 7. Kitrick Sheets provided helpful comments on the manuscript.

Development of the SIA and ACES III has been supported by the US Department of Defense’s High Performance Computing Modernization Program (HPCMP) under the two programs, Common High Performance Computing Software Initiative (CHSSI), project CBD-03, and User Productivity Enhancement and Technology Transfer (PET) and the US Department of Energy ASCR SciDAC program. The port to Blue Waters is supported by NSF under the Petascale Computing Resource Allocation (PRAC) program. We also thank the University of Florida High Performance Computing Center for use of its facilities.
REFERENCES


