An Infrastructure for Scalable Parallel Programs for Computational Chemistry

V. Lotrich, N. Flocke, M. Ponton, A. Perera, E. Deumens, R. J. Bartlett
Aces Q. C.
Gainesville, FL 32605
and
B. Sanders
Computer
Information Science and Engineering
University of Florida
Gainesville, FL 32611

October 27, 2008

UF-CISE Technical Report REP-2008-463

Abstract

Parallel implementation of complex computational software is a difficult task. This paper describes the Super Instruction Architecture (SIA) and its application to the implementation of algorithms for electronic structure computational chemistry calculations. The methods are programmed in a domain specific programming language called super instruction assembly language (SIAL), which is based on the abstractions of super instructions and super numbers. Compiled SIAL programs are executed by a parallel virtual machine known as the Super Instruction Processor (SIP). All compute intensive operations, such as tensor contractions, diagonalizations, etc, all communication operations and all input-output operations are handled by the SIP. By separating the algorithmic complexity in the SIAL from the complexities of parallel execution on computer hardware in the SIP, a software system has been created that allows for very effective optimization and tuning on different hardware architectures with quite manageable effort.

1 Introduction

Parallel implementation of algorithms is a difficult task. Ideally, we want parallel software to be efficient, making optimal use of the available computing resources; as simple as possible since easy-to-understand code can be more readily developed, debugged, verified, and modified; portable; and scalable. However, these forces conflict with each other and obtaining a good balance between them is difficult.

In this paper, we describe the Super Instruction Architecture (SIA) which was designed to support scalable parallel implementations of algorithms used in computational chemistry for electronic structure

calculations, in particular, coupled cluster methods[2]. Although certain aspects of the SIA implementation are quite specialized for this domain, we believe the general approach is more widely applicable.

Compared with other approaches to electronic structure calculations, coupled cluster methods give the most accurate results, but are very computationally intensive. In this domain, the size of the problem that can be solved is constrained by the size of the currently available machines and available computing resources. Thus in order to make the best use of the resources that are available, efficiency is important. Perhaps even more important, however, are portability and scalability, so that that as larger machines become available, they can be exploited without requiring major redevelopment effort. The algorithms in computational chemistry are extremely complex and frequently a much bigger performance gain can be obtained by improving the algorithm than by tuning its parallel implementation. Thus, in addition to portability and scalability, an environment where computational chemists can easily experiment with different algorithms is vital.

The SIA consists of two major components. The first is a domain specific programming language called SIAL (Super Instruction Assembly Language, pronounced "sail"). SIAL was designed to make it easy for computational chemists to express their algorithms in such a way that they can be easily understood, and also effectively executed in parallel. The latter is achieved by offering the abstractions of super instruction and super number which support expressing algorithms in terms of operations (the super instructions) on blocks of data (the super numbers) rather than individual floating point numbers. The runtime component, SIP (super instruction processor), is a parallel virtual machine that provides for parallel execution of the SIAL programs. The SIP manages the complexities of dealing with parallel hardware, including communication and I/O, and provides efficient implementations of commonly used computationally intensive operations from the application domain. When a new architecture becomes available, the SIP needs to be ported and tuned for the new architecture while the SIAL programs themselves will run unchanged. By separating the algorithmic complexity in the SIAL from the complexities of execution on computer hardware in the SIP, the SIA allows for very effective optimization and tuning on different hardware architectures with quite manageable effort.

The SIA has been used successfully to implement ACES III[11], the parallel successor to the widely used (sequential) package ACES II[18], which incorporates several advanced methods in electronic structure computations including coupled cluster theory. Coupled cluster theory is extremely complex and the first implementations of the energy [14, 2] and gradient [19] calculations using this method on serial computers took several man-years of effort. Several other attempts have been made to write parallel implementations [16, 3] of coupled cluster methods with varying degrees of success. In all these parallel implementations, assumptions were made during the design about the relative sizes of certain data structures which eventually imposed restrictions on the applications that could be handled. Retuning to support new and larger applications turned out to be very labor intensive. A detailed discussion of the computational chemistry methods incorporated in ACES III and survey of other available software for electronic structure methods can be found in [11].

In contrast, the SIA has proved to be quite flexible. The SIA and ACES III have been implemented on a wide range of architectures such as the IBM POWER4 cluster, the HP/Compaq SC45 cluster, the Cray X1 system, the SGI Altix, and Linux clusters with Myrinet or Infiniband interconnect, and most recently the Cray XT3 and XT4, with good performance and reasonable programming effort.

The remainder of the paper is organized as follows: First, we discuss the relevant features of the problem

domain and show how this leads to the notions of super numbers and super instructions. Then, we describe the design and implementation of SIA in more detail, first focusing on the programming language SIAL in section 3, followed by the design of the SIP in section 4. Tuning is discussed in section 5 and a comparison with alternative programming models is found in 6.

2 Overview

2.1 Characteristics of the problem domain

The goal of the computations is to provide an approximate solution Schroedinger's equation for various configurations of molecules. These computations are dominated by algebra with matrices and tensors. If implemented in C or Fortran, these data structures would be implemented using arrays with two, three, four or more indices. Most problems of interest will contain arrays that are too large to fit into the memory of a single processor, and sometimes 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 in blocks and distributed throughout the system.

Middleware and parallel programming languages that support distributed data structures in some way have long been available. Several of these are discussed in Sec. 6. While the details vary, generally, an abstraction is provided that helps to relieve the programmer from the tedious bookkeeping involved in locating and moving data in the system. Although data may be moved in larger chunks, algorithms are primarily written in terms of individual data elements.

2.2 Super numbers and super instructions

The SIA departs from th approach taken by most systems supporting distributed data by introducing the notions of *super numbers* and *super instructions*. Each dimension of an array is broken up into segments, which in turn define blocks of floating point numbers. These blocks 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 often use the term block instead of super number.) Super instructions are implemented efficiently in the SIP. In SIAL programs, loops iterate over segment numbers rather than the indices of individual elements. An iteration typically contains one or more super instructions. Thus algorithms are written in terms of super numbers and super instructions that operate on blocks of data. Individual numbers can be accessed in SIAL programs, but this should be rare as it impacts performance negatively. SIAL source programs are independent of implementation specific details such as the number of processors, the location of data, and the size of segments.

As an example, the following typical summation over two indices,

$$R_{ij}^{\mu\nu} = \sum_{\lambda\sigma} V_{\lambda\sigma}^{\mu\nu} T_{ij}^{\lambda\sigma} \tag{1}$$

could be expressed in terms of block contractions as

$$R(M,N,I,J)_{ij}^{\mu\nu} = \sum_{LS} \sum_{\lambda\in L} \sum_{\sigma\in S} V(M,N,L,S)_{\lambda\sigma}^{\mu\nu} T(L,S,I,J)_{ij}^{\lambda\sigma} \tag{2}$$

Here we have introduced two sets of indices. The indices M, N, L, S, I, J are block indices. Thus, for example, $V(M, N, L, S)$ is itself a 4-index matrix containing G^4 elements where G is the segment size, which typically would be chosen to be between 10 and 50. We define the contraction of one block of V with one block of T, which produces one block of R, as one contraction super instruction. This instruction can be implemented as a DGEMM and involves 2×100^3 to $2 \times 2,500^3$ floating point operations. The SIAL program does not access the indices inside the block; these are only visible inside the super instruction. On a 1 GHz processor, this operation takes from 2 milliseconds to 16 seconds.

The move from expressing algorithms in terms of operations on individual numbers to operations on blocks of numbers ensures that the algorithm is expressed in a way that is likely to admit an effective parallel implementation. Because blocks are the units of data that are moved around the system between processors and various levels in the memory hierarchy, by choosing an appropriate block size, the application can be tuned to make most effective use of the communication hardware and caches. The segment sizes are read from a file during program initialization, thus can easily be changed when advantageous to do so. In particular, the time to transfer a block between processors can be tuned to be less than the time for a block compute operation with a super instruction. With careful scheduling and pre-fetching of data, a large portion of all communication can be hidden behind computation, resulting in favorable parallel performance of the program. In practice, the time to execute a computation super instruction and the time to execute a memory super instruction range from similar to an order of magnitude different.

In many situations, the block algorithms look almost identical to the ones for individual numbers. In others, expressing an algorithms in blocks may require the programmer to totally rethink their algorithm. However, in these cases, the original algorithm would have performed very poorly in parallel and needed significant modification in any case. Thus the abstraction hides many details (such as the segment size, location of data, etc.) while strongly supporting the creative thinking necessary to obtain a scalable parallel algorithm.

3 The programming language SIAL

In this section, we give an overview of the most relevant parts of the SIAL programming language. A complete description can be found in [12].

3.1 Data Types for Arrays

Although the size and number of segments belonging to an array are not known precisely when the program is written, the qualitative differences between fitting in the memory of a single processor, fitting in the collective memory of the system, and residing on disk is exposed to the programmer by offering several different types of arrays. These were chosen based on an analysis of the computational needs of the domain and experience obtained during the initial development of the SIA. SIAL offers the following array types:

- **Static array** A static array is small enough to fit easily into the memory of a single processor and is replicated across all of the processors in the parallel computation. The two-index transformation matrix from atomic to molecular orbitals is an example of a static array.

- **Temporary array** A temporary array provides locally accessible copies of blocks belonging to a distributed or served array. These blocks are allocated when used, and are deallocated when they go out of scope at the end of a loop or procedure.

- **Local array** A local array is a special case of a temporary array in which one or more of the indices is formed completely. It is used like a temporary array but adds flexibility to the algorithm implementation.

- **Distributed array** Distributed arrays fit into the combined memory of a reasonable number of processors and are explicitly allocated and deallocated with `create` and `delete` operations, allowing space to be reused for different distributed arrays in different sections of the program. To read or write a particular block of a distributed array, `get` and `put` instruction are used. For example, `get v3(p,q,r,s)` brings the block indicated by the current values of `p`,`q`,`r`, and `s`, to the current processor and `put v3(p,q,r,s) = v2(p,q,r,s)` copies the indicated block of local array `v2` to the distributed array `v3`, replacing the existing block of `v3`.

  An example of a distributed array is the four-index array with transformed two-electron integrals in the molecular orbital basis containing four occupied indices V_{kl}^{ij}. For calculations with 100 and 200 occupied orbitals, V_{kl}^{ij} requires 0.8 Gbyte and 12.8 Gbytes, respectively. This amount of data can be easily distributed over a relatively small number of processors.

- **Served array** Served arrays are stored on disk and their blocks served to any processor when needed. Served arrays are necessary for arrays too large to fit into distributed memory, such as the transformed integrals array with three unoccupied indices and one occupied index. With 100 occupied orbitals and 500 basis functions, there are 400 unoccupied orbitals and then V_{ic}^{ab} takes 51 GB; with 200 occupied orbitals in a basis of 1,000 functions, it takes 820 GB.

3.2 Scalar data types and predefined constants

SIAL provides several scalar types and predefined constants that take values of these types.

- **scalar** A scalar variable holds a floating point value and is local to a task.

- **index** An index variable is a subrange of the integers that declared along with its range and used in sequential iteration.

- **aoindex**, **moindex**, **moaindex**, **mobindex**, and **laindex** These type are used for variables that index segments (not individual values) of distributed and served arrays. They are declared with their ranges. These are all a subrange of integers, but are given types that have semantic meaning in the domain[1]. Integrating this domain knowledge into the SIAL type system allows the SIAL compiler to check for consistent usage and provides information that can be exploited by the SIP.

SIAL includes a set of predefined names that are either a scalar, an index type, or a static array. These are meaningful in the domain and their values are obtained from a file when the SIAL program is initialized.

[1]The first four indices stand for types of orbitals: atomic orbital, molecular orbital, molecular orbital with α spin, and molecular orbital with β spin. An `laindex` is not associated with a particular orbital type; it is used for linear algebra operations.

Recall that the indices must be given when an array is declared, and the range must be specified when a variable of one of the index types is declared. Predefined constants are often used in these declarations. As a result, the index ranges are not necessarily hard coded into the program, and can be easily changed as needed for any execution of a SIAL program. Nevertheless, they are fixed for the duration of the computation, and guaranteed to satisfy certain domain specific constraints on their relative magnitudes, making it easier for the SIP to determine effective resource allocation and scheduling.

3.3 Control structures

In addition to procedures and an alternative statement (`IF-ELSE-ENDIF`) with the obvious meaning. SIAL provides two loop constructs.

- **D0-ENDDO** The `DO-ENDDO` construct is parameterized by a variable of type index and provides standard serial iteration over the range of the index variable.
- **PARDO-ENDPARDO** The `PARDO-ENDPARDO` is given a list of variables, each one of the index types, and iterates over all possible combinations of the values in the ranges of the given indices. The PARDO loop is subject to the following constraints
  - Dependencies between data used or produced in the loop body are not allowed, thus the iterations may be done in parallel.
  - PARDO loops may not be nested

  The SIAL programmer has no control over how this loop is executed, all scheduling is done by the SIP.

  The PARDO loop is the only way to specify parallelism in a SIAL program. As a result, properly synchronized SIAL programs exhibit weak sequential equivalence[2]. This simplifies reasoning about the correctness of SIAL programs.

3.4 Super instructions

A SIAL programmer has a collection of super instructions at his or her disposal. Although the set of super instructions is extensive, additional super instructions can be added to those offered by the SIP without changing the SIAL language itself. As discussed earlier, super instructions are implemented in the SIP and perform the computational tasks on blocks that make up the bulk of a computation.

In addition to these computational super instructions, many of which are domain specific, super instructions are also provided for a variety of operations that can be performed by the SIP. These include I/0, barriers, and a variety of utility functions.

Two types of barrier super instructions are provided, one that provides a barrier among the worker tasks that are handling the computation, and another that provides a barrier for the I/O tasks. Generally, a barrier is required between conflicting[3] super instructions.

[2] A program exhibits weak sequential equivalence if it yields the same results, except for differences due to reordering floating point operations that are mathematically associative and commutative, regardless of the number of processors on which it is executed

[3] Two super instructions conflict if they access the same distributed or served array and at least one is a write

The super instruction, `blocks_to_list` serializes distributed arrays to disk in a way that another super instruction, `list_to_blocks` can read and reconstruct the distributed array. 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, for some reason, the program is unable to run to completion.

3.5 Example

In this section, we show a brief extract from a SIAL program to give the flavor of the language. Consider the representative term in the CCSD[4] electronic structure method that was shown in equations 1 and 2. This expression can be coded in SIAL as shown below.

```
pardo M,N,I,J
  tempsum(M,N,I,J) = 0.0
  do L
    do S
      get T(L,S,I,J)
      compute_integrals V(M,N,L,S)
      temp(M,B,I,J) = V(M,N,L,S) * T(L,S,I,J)
      tempsum(M,N,I,J) += temp(M,N,I,J)
    enddo S
  enddo L
  put R(M,N,I,J) = tempsum(M,N,I,J)
endpardo M,N,I,J
```

The `pardo A,B,I,J` statement indicates parallel execution of the loop over the segment indices `A,B,I,J`. The ranges of the indices were given when the index variables were defined elsewhere in the complete program. Each task will perform the iterations for the index values that have been allocated to it. The array `tempsum` will be used to accumulate the results of the summation and is local to the task. Its size is determined by the ranges of the indices and all of its elements are initialized to 0.0. The statements `do L` and `do S` are sequential loops over the complete ranges of `L` and `S`. The statement `get T(L,S,I,J)` obtains the given block of the distributed array T. If the block happens to be stored at the processor, then the statement does nothing, otherwise it initiates an asynchronous communication request to obtain it from whichever processor holds it. Recall that V was the 8 TB array containing the 2-electron integrals. Since the entire array is too big to store, each block is computed on demand using the built-in function `compute_integrals`. If the preceding `get` statement requires communication, it will be overlapped with the integral computation. In the next two statements, the contraction of a block of `T` and a block of `V` is computed and stored in local array `temp`, while `tempsum` accumulates the sum. The implementation of the contraction operator, '*', ensures that the necessary blocks are available and waits for them if necessary. The `put` statement saves the result to a distributed array, `R`. Like the `get` instruction, a `put` instruction may require communication with another processor if the indicated block has not been assigned to the current task. The runtime system, SIP, handles the data allocation, communication, and locking necessary to properly implement the SIAL semantics.

[4]coupled cluster (CC) theory for single and double excitations

4 SIP

The SIP (super instruction processor) is a parallel MIMD virtual processor and run-time environment written in C and FORTRAN with a master task, a set of worker tasks, and a set of I/O server tasks. We have developed implementations where SIP is a pure MPI program, where it is a mixed MPI POSIX threads program, and where it uses SHMEM for communication. At this time, the pure MPI program implementation performs best on all platforms.

4.1 SIAL object code

A SIAL program is compiled into super instruction object code which is interpreted by the SIP. The object code includes tables with definitions of all variables and arrays, including index ranges, and the list of super instructions to be executed. The main table is the instruction table with instruction codes and operand addresses. The operand addresses are entries in data descriptor tables. A data descriptor for a distributed or served array contains the location of each block in the array. Some instructions are the compute intensive instructions that we normally think of as super instructions. Others are control instructions for loops, selection, etc. The super instructions corresponding to `get`, `put`, `prepare` and `request` commands may also initiate communication. As much as possible, the super instructions are executed asynchronously: Communication operations are started and then control returns to the SIP task so that more computations or different communications can be performed. When the instruction that needs the data starts executing, it first checks that the communication instruction has completed successfully and it will wait if the communication is still in progress.

4.2 The master and worker tasks

When a SIAL program is executed, the master task performs the management functions required to set up the calculation. In order to gather information about the memory usage in the program, the master task runs through the SIAL program in a dry-run mode, essentially executing the program without performing communication, I0, or compute intensive operations. Using the gathered information, the master sets up the data distribution by deciding which blocks of each declared distributed array will be owned by each worker task, and which I/O server task is responsible for each block of a served array. The tables containing this information, which remain fixed throughout the computation, are replicated at all worker tasks. As a result, a worker needing a block of data can immediately determine the location from which to request it. After the initialization, the master task becomes a worker task and participates in the parallel computation and holds part of the distributed data.

Although not obvious at first glance that this would be the case, a simple, fixed data placement strategy, even with no attempt to place data at the task most likely to need it, works well in practice. There are two main reasons for this. First, the applications tend to exhibit irregular access patterns with little spacial or temporal locality. Also, recall that the segment size can usually be chosen so that much of the communication is overlapped with computation in any case. If the algorithm is not overlapping communication with computation effectively, then solving this problem will make a much more significant difference in the performance than would more elaborate schemes to improve the data placement.

The memory in each SIP task is divided into several stacks of blocks. Data blocks can have 2, 3 or 4

indices, their size thus ranges from S^2 to S^4, where S is the segment size. Arrays with more indices can be defined, but additional indices are simple indices familiar from C and Fortran with a segment size of 1. A typical segment size could be 10. It is clear that it would be wasteful to store a 2-index block, *e.g.* of size 100 floating point numbers, in a universal block container that can accommodate 4-index blocks, *e.g.* of size 10,000. For this reason the SIP supports several block stacks with varying block container size. To get optimal performance, it is important to make sure the block size has the correct relation to the cache of the microprocessor used.

Recall that parallelism is specified by the SIAL `PARDO` instruction. In SIP, the iterations for the `PARDO` are divided among the worker tasks by the master. Initially, the work is divided into "chunks" and doled out to the worker tasks. When a worker completes its chunk, it requests another chunk from the master. The chunk size decreases exponentially as the computation proceeds. This is similar to the approach taken with "guided" scheduling in OpenMP and results in a good load balance.

Super instructions are provided for computational tasks in the domain. For example, there is a set of super instructions to perform various tensor contractions between blocks. The super instructions differ in the possibilities of which index of the first operand is combined with which index of the second operand and which index to map into which result index. If the super instruction operates on a block, the SIP determines whether the block is local; if it is not, then a request is made from the owner task of the block. The SIP may look ahead and request several blocks that it expects will be needed in the next iteration of a loop. It can start a multiplication operation for which all operands are available. The SIAL programmer does not know where the block resides and does not need to worry about it. If the programmer wants to ensure that only local blocks are generated and used inside a loop, then, for example, a temporary array can be used for the intermediate results.

4.3 I/O Server Tasks

The I/O server tasks are dedicated to providing support for the served arrays defined in SIAL. Blocks are written to the served array with the `prepare` instruction. The worker task looks up the responsible I/O server task and sends the block. I/O server tasks have a cache where blocks of served arrays can be held. When a block is prepared, it is placed in the cache and written to disk lazily. When a worker task needs a block from a served array, it sends a `request` to the responsible I/O server. If the block has recently been prepared or has been prefetched, it is likely to be in the cache and can be returned directly to the requesting worker without waiting for disk I/O. To facilitate prefetching, the request instruction in SIAL contains a parameter indicating which index will be incremented next in the loop.

All operations of the I/O server task are non-blocking: all communication uses `mpi_isend` and `mpi_irecv` and all input-output uses asynchronous read and write. This prevents a single operation from blocking progress on an independent operation. Another feature of the implementation is that no blocks are explicitly allocated in any I/O server task block pool or on any hard disk drive for declared served arrays. Instead, a block is allocated when it is filled with data, which is done with the SIAL `prepare` instruction. This allows the SIAL programmer to easily exploit symmetry in data structures: a served array can be declared as usual, the algorithm would use a knowledge of symmetry to refer to, say V(i,j) instead of V(j,i). There is no cost to declaring V with a declaration for a normal block-shaped structure.

At a low priority, every task in the I/O company investigates the status of all data blocks in its block

cache, and when the block pool fills up above some threshold, the I/O server task copies blocks to locally accessible disk storage using an LRU replacement strategy. Blocks that are often read or changed regularly will remain in memory for quick access, whereas blocks that are used infrequently migrate to disk. If a request for a block is made that is not resident in memory, a delay will occur before the request completes because the block must be restored from disk first.

An IO server tasks may also perform certain computations, namely computing blocks of integrals, which are important in the problem domain. [5] If a request arrives for a block and it is not found in the block pool of the I/O server responsible for the block or on its hard disk drive, the I/O server task assumes that an integral block is needed, and it starts an integral computation, which will be transmitted when the computation completes.

5 Performance tuning

One of the design goals of the parallel implementation of ACES III was to be able to perform significant tuning for different platforms without the need for a major rewrite of code every time. The SIA organizes the execution of the parallel program into atomic steps, namely the execution of one super instruction, that all take a finite amount of time to complete. As a result it is possible to include time-statistics gathering in each super instruction without noticeable impact on application performance.

In our experience, an effective way to create a high performance SIAL program is to write an initial implementation of the algorithm without worrying too much about performance. After the program is validated to be correct, the timer information is used to show the places where performance tuning work is needed. The timer output is organized to print the average time, minimum, maximum and standard deviation over the number of processors next to each instruction in an annotated source code listing. The ones to look at first are the block wait times, the PARDO loop times, and then the times for instructions inside the PARDO. The block wait time indicates that loops or instructions where called but could not proceed because some required block of data had not arrived yet. In most cases the problems will show up on one or more PARDO loops. Sometimes a performance problem is caused by something that happens in code before a PARDO loop, but in most cases the timers for the instructions inside a PARDO loop will clearly point to what needs to be modified to improve performance.

In Fig. 1 the timing for an MBPT(2) gradient calculation on Cytosine+OH is displayed before and after performance tuning. The calculation has 67 correlated occupied orbitals and 552 basis functions. Calculations were done on 14, 32, 64, 128, and 256 processors on a SUN Linux cluster with InfiniBand interconnect. Observe that scaling is good in both cases, but absolute performance is improved by a factor 4.76 (from 681 min to 143 min) for 32 processors to a factor 3.71 (from 89 min to 24 min) for 256 processors.

[5] In electronic structure computations the partial differential equation being solved are defined in $2n$ dimensional space, with n the number of electrons. Discretization of the problem on a grid, so successful in problems in mechanical engineering and computational fluid dynamics, is not practical. The problem is discretized using three dimensional basis functions, mostly Gaussian type functions. The action of the operators in the partial differential equations then leads to actions on these basis functions. The fundamental Hamiltonian in electronic structure theory turns out to only need operators acting on a basis function or on pairs of basis functions. Thus a three dimensional integration or a six-dimensional integration will lead to a matrix representation of the $3n$ dimensional partial differential equation. The matrix elements are thus expressed in terms of one-electron and two-electron integrals, which must be computed and manipulated in any algorithm used to obtain (approximate) solutions to these equations.

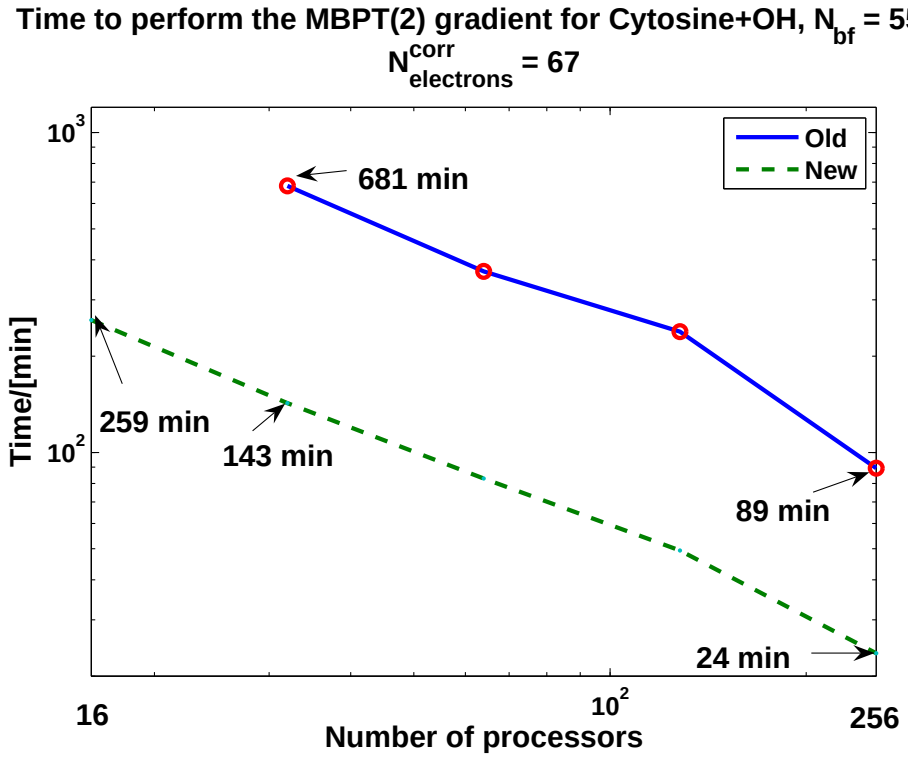

Figure 1: (Color online) Scaling of RHF MP2 gradient of Cytosine+OH before and after the performance tuning code changes (67 correlated ocupied orbitals, 552 bf) calculations for 16, 32, 64, 128, and 256 processors.

Analysis of the timer report from running the initial SIAL program, pointed to three problems:

1. Some tensor-contraction super instructions were expensive compared to other. These instructions have internal intelligence that allows them to recognize when the data as given is stored in a sub-optimal way that will result in bad strides in inner loops and concomitant cache misses. In most cases, the execution is optimized by performing transpositions or other operations on one or both operand blocks. This computation, however, revealed some rare cases that were not optimal, leading to an improved implementation of the super instruction. In addition to optimizing inside the instruction, it is possible to present the transposed matrix to the super instruction, thus replacing multiple data sorting steps inside every instruction by a single one outside a loop. The transposed array was stored in a short-lived temporary array. This optimization is straightforward and contributed to the performance improvement for this program.

2. The accumulation of some small arrays appeared too expensive. A better way to write the operation reduced the number of operations by creating better intermediate quantities in short-lived, partial four-dimensional local arrays.

3. Permutations of data to make use of the symmetry of two-electron integrals so as to avoid having to recompute them, was not done at optimal times. The changes also including better placement of these permutations in the code.

The program that was optimized in this examples consists of 6,000 lines of SIAL code. The optimization

was completed in two days by one programmer.

6 Related Work

It is important to point out that we feel that the SIA based design is not a totally new development. Rather, it is a natural evolution of what has been developed in that past two decades. It is worthwhile to exhibit how SIA relates to several of the existing development tools for parallel implementation, particularly in the domain of electronic structure calculations in computational chemistry.

There are two related language developments, Co-array Fortran[7] and UPC [5], that have gained much attention recently for enabling faster development of parallel applications. Both languages take the approach of extending a general purpose programming language to support a Partitioned Global Address Space model, where data can be either shared or local, and shared data is assigned an affinity to a particular processor. The notion of affinity can be used to improve the efficiency of algorithms by exploiting locality.

Although the syntax is very intuitive and simple, it requires programmers to be rather explicit about how the data is laid out. This will build in assumptions about data locality into the program, which in turn makes it likely that the design of the program will have to be revisited when it is ported to new hardware with significantly different characteristics. It is not clear that language features providing easy access to distributed data will allow petascale systems to be programmed efficiently for the majority of high performance computing problems.

The Global Arrays (GA) toolkit[8, 13] developed for NWChem[15] and used in MOLCAS [9] and COLUMBUS[10] provides a programming interface to distributed data structures for an MIMD program that supports arrays similar to Co-array Fortran and UPC. The regularity of the distribution of the data over the cooperating tasks in the MIMD program has advantages and disadvantages. The performance can be optimized by using the available information about the regularity of the data structure. But the regularity imposes limitations when trying to run applications with a smaller number of tasks: Even though the total size of distributed memory is sufficient to hold the data, the imposed regular structure requires more tasks to allow the MIMD program to run. The SIA in contrast imposes the structure of blocks, but the blocks can be located anywhere with very little regularity. The master task does try to distribute data blocks evenly to optimize performance, but this is not known or controlled by the SIAL program and may be changed by the SIP or by the user at runtime. This extra flexibility allows SIAL programs to run on more configurations than GA programs with similar problems.

The SIA offers a range of arrays that allow programmers to efficiently express varying needs for data distribution and data access patterns. 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, which is unlike what is done in Co-array Fortran, UPC, and GA, where the specification is precise and complete. In SIA a significant part of the data layout is left to the SIP and some is left even the the end-user of the application.

The Distributed Data Interface (DDI) [6] of GAMESS [17] is simpler in design than GA, but also is very specific in how the data is laid out. With the recent extension to optimize the use of shared memory nodes[4], 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. Then the end user is confronted with the situation where the program allocates data in a way that does not match the available computer system resources (number

of CPUs and memory) with the result that the calculation will not run. The SIA has fewer limitations like this because of the extra flexibility provided by SIP in making data-layout decisions at runtime.

We note that the SIA design is similar to that advocated in [1] 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.

7 Conclusion

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

1. The SIAL language is not as general as Fortran and C/C++, but it is very expressive and easy to read and write. We have found that the balance of expressiveness and hiding of detail it provides creates a very productive environment. Our programmers have been able design, code and debug the implementation of numerous algorithms in record time.

2. Because all basic operations are relatively large, we can keep track of very detailed performance metric throughout every execution without an impact on performance. This provides the programmers, both SIAL and SIP, with very detailed information about where the program spends its time. This has lead to many optimizations, some in SIAL, some in SIP. An optimization of the SIP requires no change at all in any of the SIAL programs.

3. Because SIP has been designed to be fully asynchronous, it has been very easy to port to different architectures. We started with IBM SP systems, ported to HP-Compaq, Cray X1, SGI ALtix, Cray XT3, Linux clusters. Performance varies in absolute terms, but on each platform the performance and scaling are quite good with very little tuning of SIP and no change in any SIAL programs. The ports required switching from MPI with POSIX threads to shmem or to pure MPI. These changes were confined to small sections of code and were quite painless.

4. The ease of writing algorithms in SIAL has prompted our SIAL developers to often write multiple implementations of the same algorithm to use the two versions as tests of each other. Such development takes days, rather than weeks typical for development in Fortran or C/C++ with explicit MPI and POSIX threads calls.

The design with SIA has also clearly identified the computationally intensive kernels and has 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 attached processors, and field programmable gate arrays (FPGA).

The way that the interface between SIAL and the SIP was designed has turned out to be very effective. Based on our experiences so far, we have identified opportunities to enhance SIAL and also provide useful tool support for SIAL programmers. We have also identified areas where the SIP can be improved, especially to support scaling to petascale. To a significant extent, the future work on SIAL, and on the SIP will be able to proceed independently.

It is not our claim that SIA based design is a completely new development, rather we feel it is a natural evolution of the work that has been done by many before us in the past two decades. We have carefully studied what has been tried and have built on what was found to work.

Acknowledgments

The development of ACES III was largely supported by the US Department of Defense's High Performance Computing Modernization Program (HPCMP) under two complementary programs: Common High Performance Computing Software Initiative (CHSSI), project CBD-03, and User Productivity Enhancement and Technology Transfer (PET). We also thank the University of Florida High Performance Computing Center for use of its facilities.

References

[1] Krste Asanovic, Ras Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John D. Kubiatowicz, Edward A. Lee, Nelson Morgan, George Necula, David A. Patterson, David A. Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine A. Yelick. The parallel computing laboratory at u.c. berkeley: A research agenda based on the berkeley view. Technical Report UCB/EECS-2008-23, EECS Department, University of California, Berkeley, March 2008.

[2] R. J. Bartlett and M. Musial. Coupled cluster theory in quantum chemistry. *Rev. Mod. Phys.*, 79(1):291–352, January 2007.

[3] G. Baumgartner, A. Auer, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. J. Harrison, S. Hirataand S. Krishnamoorthy, S. Krishnan, C. Lam, Q. Lu, M. Nooijen, R. M. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. Synthesis of high-performance parallel programs for a class of ab-initio quantum chemistry models. *Proceedings of the IEEE*, 93(2):276–292, 2005. Special Issue on program generation, optimization and platform adaptation.

[4] Jonathan L. Bentz, Ryan M. Olson, Mark S. Gordon, Micheal W. Schmidt, and Ricky A. Kendall. Coupled cluster algorithm for networks of shared memory parallel processors. *Comput. Phys. Comm.*, 176:589–600, 2007.

[5] Unified Parallel C. http://upc.gwu.edu/ and http://upc.lbl.gov/.

[6] Graham D. Fletcher, Micheal W. Schmidt, Brent M. Bode, and Mark S. Gordon. The distributed data interface in gamess. *Comp. Phys. Comm.*, 128:190–200, 2000.

[7] Co-Array Fortran. http://www.co-array.org/.

[8] R. J. Harrison. Global arrays. *Theoret. Chim. Acta*, 84:363–375, 1993.

[9] G. Karlström, R. Lindh, P.-A. Malmqvist, U. Ryde, V. Veryazov, P.-O. Widmark, M. Cossi, B. Schimmelpfennig, P. Neogrady, and L. Seijo. Molcas: A program package for computational chemistry. *Comp. Mat. Sci.*, 28:222–239, 2003.

[10] Hans Löschka, Ron Shepard, Russell M. Pitzer, Isaiah Shavitt, Michal Dallos, Thomas Muüller, Péter Szalay, Michael Seth, Gary S. Kedziora, Satoshi Yabushita, and Zhiyong Zhang. High-level multireference methods in quantum-chemistry program system columbus: analytic mr-cisd and mr-aqcc gradients and mr-aqcc-lrt for excited states, guga spin-orbit ci and parallel ci density. *Phys. Chem. Chem. Phys.*, 3:664–673, 2001.

[11] V. Lotrich, N. Flocke, M. Ponton, A. D. Yau, A. Perera, E. Deumens, and R. J. Bartlett. Parallel implementation of electronic structure energy, gradient and hessian calculations. *J. Chem. Phys.*, 128:194104 (15 pages), 2008.

[12] Victor Lotrich, Mark Ponton, Erik Deumens, and Rod Bartlett. *ACES III SIAL Programmer Guide*. University of Florida, Gainesville, Florida, 2006.

[13] I. Nieplocha, R.J. Harrison, and R. J. Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. In *Proceedings of Supercomputing 1994*, page 340, Washington D.C., 1994. IEEE Computer Society Press.

[14] G.D. Purvis and R.J. Bartlett. A full coupled-cluster-singles and doubles model: The inclusion of disconnected triples. *J. Chem. Phys.*, 76:1910–1918, 1982.

[15] Kendall R.A., Apra E., Bernholdt D.E., Bylaska1 E.J., Dupuis M., Fann G.I., Harrison R.J., Ju J., Nichols J.A., Nieplocha J., Straatsma T.P., Windus T.L., and Wong A.T. High performance computational chemistry: An overview of nwchem a distributed parallel application. *Comput. Phys. Comm.*, 128(1):268–283, June 2000.

[16] A.P. Rendell, T.J. Lee, and A. Komornicki. A parallel vectorized implementation of triple excitations in ccsd(t): Application to the binding energies of alh3, alh2f, alhf2, and alf3 dimers. *Chem. Phys. Lett.*, 178:462–470, 1991.

[17] M. W. Schmidt, K.K. Baldridge, J. A. Boatz, S.T. Elbert, M. S. Gordon, J. H. Jensen, S. Koseki, N. Matsunaga, K. A. Nguyen, S. Su, T. L. Windus, M. Dupuis, and J. A. Montgomery. Gamess quantum chemistry software. *J. Comput Chem.*, 14:1347–1363, 1993.

[18] J.F. Stanton, J. Gauss, J.D. Watts, W.J. Lauderdale, and R.J. Bartlett. ACES II Program System. University of Florida, 1994.

[19] J.D. Watts, J. Gauss, and R.J. Bartlett. Coupled-cluster methods with non-iterative triple excitations for restricted open-shell hartree-fock and other general single determinant reference functions. energies and analytical gradients. *J. Chem. Phys.*, 98:8718–8733, 1993.