Refactoring a Language for Parallel Computational Chemistry

Beverly A. Sanders
Department of Computer and Information Science and Engineering, University of Florida
sanders@cise.ufl.edu

Erik Deumens
Quantum Theory Project, University of Florida
deumens@qtp.ufl.edu

Victor Lotrich
ACES QC, Gainesville, FL
lotrich@qtp.ufl.edu

Mark Ponton
ACES QC, Gainesville, FL
ponton@qtp.ufl.edu

Abstract
We describe a project to provide refactoring support for the SIAL programming language. SIAL is a domain specific parallel programming language designed to express quantum chemistry computations. It incorporates language support for the loop parallelism and distributed array parallel design patterns. In contrast to refactorings typically undertaken for object-oriented programs which have the goal of improving the code structure, SIAL refactorings are usually done to improve the performance or to allow larger problems to be solved.

Categories and Subject Descriptors CR-number [subcategory]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction
In this paper, we describe our initial efforts to provide refactoring support for the SIAL programming language. SIAL (Super Instruction Assembly Language, pronounced "sail") is the productivity layer of the SIA (Super Instruction Architecture) which was designed to support scalable parallel implementations of algorithms used in computational chemistry for electronic structure calculations, in particular, coupled cluster methods [BM07]. The syntax of SIAL is close to the mathematical notation used by scientists in this area and its programs are easy for the computational chemist to write, and no less importantly, to read. One of the main ideas of the programming model supported by the language is that application programmers should (most of the time) express computations in terms of blocks of data rather than individual floating point numbers. This has two consequences: data is normally handled at a granularity that can be efficiently moved around, and computations steps (operations on blocks) will be time consuming enough for the runtime system to be able to automatically overlap communication and computation effectively. In addition, the language exposes qualitatively different implementations of arrays, whether they are local to a task, distributed over multiple tasks, or residing on disk.

The efficiency layer of the SIA is the SIP (Super Instruction Processor) which is a parallel virtual machine written in C, Fortran, with MPI that takes care of allocating and locating blocks, managing communication, performing I/O, and other lower level details. SIAL programs are efficient and scalable. 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 already been implemented for several different systems and it has been used to implement the ACES III [LFP+08] computational chemistry software package.

The support for refactoring is being implemented as part of an Eclipse [Ecl] based IDE for SIAL. The parser and scanner were developed using the LALR parser generator LPG [LPG], and the IDE development utilizes the IMP IDE metatooling platform [IMP]. This work is part of a larger project to evaluate and further enhance the language SIAL, its runtime system SIP, and the development environment in order to support scaling to petascale, thus enabling the discovery of new scientific results. Another goal is to codify the general approach so that the approach and its tools can be transferred to other scientific domains.

2. Overview of SIAL

The complete language definition is found in the SIAL Programmer Guide [LPDB06]. Here, we will show a brief extract from a SIAL program to give the flavor of the language. Consider one representative term in an electronic structure calculation. The term looks like a matrix multiplication with a contraction over two indices.

\[ R_{ij}^{ab} = \sum_{cd} V_{cd}^{ab} T_{ij}^{cd} \]  

(1)

To get an idea of the amount of data to be processed, if we model a molecule with 200 electrons described by 1,000 basis functions, the two-electron integrals represented by \( V \) in the formula take up 8 TB. The \( T_{ij}^{cd} \) array contains \( 2.56 \times 10^{10} \) floating point numbers or 200 GB of data. Clearly, this problem cannot be solved without distributing blocks of the arrays over multiple processors. For this problem, it works well to divide the data into blocks by dividing all index ranges into segments of length 10 to 50. Then blocks of \( T \) and \( V \) each contain 10,000 to 6,250,000 floating point numbers, or 80 KB to 50 MB. The term can be blocked as follows

\[ R(A, B, I, J)_{ij}^{ab} = \sum_{CD} \sum_{cC} \sum_{dD} V(A, B, C, D)_{ij}^{ab} T(C, D, I, J)_{ij}^{cd} \]  

(2)

where the capital letters \( A, B, C, D, I, J \) indicate block indices and the lowercase letters \( a, b, c, d, i, j \) indicate indices inside each block ranging from 1 to 10 or 50. For example, \( V(A, B, C, D) \) is itself a 4-index matrix of size \( 10^4 \) to \( 50^4 \). This expression can be coded in SIAL as shown below.

```sial
pardo A,B,I,J
  tempsum(A,B,I,J) = 0.0
  do C
    do D
      get T(C,D,I,J)
      compute_integrals V(A,B,C,D)
      temp(A,B,I,J) = V(A,B,C,D) * T(C,D,I,J)
      tempsum(A,B,I,J) += temp(A,B,I,J)
    enddo D
  enddo C
  put R(A,B,I,J) = tempsum(A,B,I,J)
endpardo A,B,I,J
```

The pardo \( A, B, I, J \) statement indicates parallel execution of the loop over the indices \( A, B, I, J \). The ranges of the indices have been defined elsewhere in the complete program. Each task will perform the iterations for the index values that have been allocated to it. \( tempsum \) 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 initialize to 0.0. do C and do D are sequential loops over the complete ranges of C and D.

get \( T(C,D,I,J) \) obtains the given block of the distributed array \( T \). If the block is local to 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 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 SIP handles the data allocation, communication, and locking necessary to properly implement the SIAL semantics.

Super instructions are available for common operations performed in the domain (and can be added without modifying SIAL), thus raising the level of abstrac-
tion of the language from the point of view of the programmer. The \texttt{COMPUTE\_INTEGRALS} instruction in the code fragment above is an example. SIAL does not provide access to low level information like the number of processors; knowledge of hardware is restricted to the SIP.

3. Refactoring SIAL

In object-oriented programs, refactoring is typically done to improve a program’s structure without changing its behavior. In SIAL, some refactorings may be done for structural reasons, but most have the goal of either improving the performance or to make a larger problem feasible. These refactorings typically involve a tradeoff. At the least, the program becomes less straightforward and more difficult to read and understand. In many cases, in improvement in one dimension, say execution time, may come at the cost of less desirable (or unacceptable) behavior in another, such as memory usage or communication requirements.

In practice, SIAL refactoring is usually carried out in small steps and driven by empirical measurements. The output of a SIAL program execution includes statistics about the execution time and blocking time of each loop, allowing the programmer to easily identify and focus on the problem areas. The ability to conveniently display performance information, and perhaps incorporate it into program analysis is an eventual goal.

Determining what kinds of program transformations are likely to be useful enough to codify as a refactoring requires a significant amount of programming experience in SIAL, thus this project is a collaboration between computer scientists and the experienced computational chemists who use SIAL as a tool in their work. Indeed, we view the provision of a specific refactoring as one way of communicating expert advice to programmers.

Refactorings for SIAL can be roughly classified into three types: Loop improvement, structural, and data access modifications.

In this section, we will first give a detailed description of an important SIAL loop improvement refactoring. Then we briefly discuss other refactorings that we have implemented, or plan to implement.

3.1 Loop improvement

Recall that SIAL provides two types of loops. \texttt{do} specifies a sequential loop. \texttt{pardo} results in its iterations being executed in parallel and is the only way to specify parallelism in SIAL. The \texttt{pardo} can be considered programming language support for the Loop Parallelism parallel design pattern described in [MSM04]. Several of the transformations suggested in that pattern can be implemented as refactorings for SIAL. Our first example is a SIAL specific version of the idea of moving an invariant expression out of a loop.

Elminating redundant transpose operations Consider the following code fragment where \texttt{V} is a served array stored on disk, \texttt{t} and \texttt{L} are temporary arrays that hold blocks of distributed and served arrays in local storage.

\begin{verbatim}
served V(a2,a,a1,i)
...
DO i
...
t(a,i) = V(a2,a,a1,i1)*L(a1,i,a2,i1)
...
ENDDO i
\end{verbatim}

The tensor contraction operation, $*$, finds the common indices in the two four-dimensional operands $V(a2,a,a1,i1)$ and $L(a1,i,a2,i1)$, $a1$, $a2$, and $i1$, and returns a two dimensional result indexed with the remaining indices, $a$, and $i$. Although not always necessary, in this particular contraction, both operands are internally transposed so that the matching indices appear in the same order in both. In particular, $V(a2,a,a1,i1)$ is transposed to $V(a,a2,a1,i1)$, and $L(a1,i,a2,i1)$ to $L(i,a2,a1,i1)$. Then a DGEMM operation is invoked with the transposed matrices. Although it isn’t immediately obvious from the source code which was written in a straightforward way based on the original mathematical formulas, this contraction performs a non-trivial amount of redundant computation by repeating the same transpose of $V(a2,a,a1,i1)$ each time through the loop. Explicitly transposing $V$ outside the loop will eliminate the redundancy at the cost of making the program slightly less readable and increasing the amount of memory required for the loop by introducing a new temporary array. (Recall that the size of a block of a four-dimensional array can range from 80 KB to 50 MB.) To support this type of optimization of the source code, the SIAL refactoring tool provides two facilities, an editor action, and a refactoring action.
• Show suboptimal contractions This action, when activated by the programmer, traverses the AST and identifies the arrays that are implicitly transposed in a contraction and where the transposition can safely be moved out of the loop. The programmer is thus relieved of a good deal of tedious inspection of indices. The programmer considers the measured performance of the loops containing the contractions, and the memory usage situation to decide which ones to modify. If the modification is expected to have only a small influence on the performance, it is better to leave the code in its original, simpler form.

• Move transpose before loop To perform this refactoring, the programmer selects the array to be replaced and activates the refactoring command. The refactoring command checks to ensure that the selected array is part of a contraction expression, will be implicitly transposed in that context, and that none of its indices match the index variables of the enclosing loop. This will always be the case if the selected array was one indicated by the Show suboptimal contractions command. The refactoring then suggests a transposition and a name for a new temporary array. The suggested transposition is the one that would have been done internally by the contraction operation. Both the transposition and the name can be modified: if the variable name has been modified, it is checked that the name has not already been declared. If the contraction appears in one alternative of an if statement, the programmer will be asked if identical occurrences on other branches should also be modified. The actual code transformation involves introducing a declaration of the new temporary array, introducing an assignment statement to initialize the temporary array, and replacing the original array in the contraction with the transposed one. Thus, the example code would become something like

\[
\text{served V}(a2,a,a1,i) \\
\text{temp VTgen}(a,a2,a1,i) \\
\ldots \\
\text{VTgen}(a,a2,a1,i1) = V(a2,a,a1,i1) \\
\text{DO i} \\
\ldots \\
\begin{aligned}
  \text{t}(a,i) &= \text{VTgen}(a,a2,a1,i1) \ast \text{L}(a1,i,a2,i1) \\
  \ldots \\
\end{aligned} \\
\text{ENDDO i}
\]

Note that the assignment to VTgen above implicitly transposes the data in the block rather than simply copying it. Also, in SIAL, but not in more expressive languages, it is sufficient to simply compare the names of indices in order to determine if an array block depends on the loop index.

• Move transpose after loop In the Move transpose before loop refactoring described above, we move the transposition of an argument to the contraction before the loop. It is sometimes desirable to store the results of a contraction in a way that admits a more efficient contraction operation and then transpose it after the loop.

• Merge loops Two subsequent loops over the same indices can be merged to a single loop.

• Exchange loops For example,

\[
\begin{aligned}
&\text{do C} \\
&\text{statements mentioning C} \\
&\text{pardo A,B} \\
&\ldots \\
&\text{endpardo A,B} \\
&\text{statements mentioning C} \\
&\text{endo C} \\
&\text{is transformed to the better performing} \\
&\text{pardo A,B} \\
&\text{do C} \\
&\text{statements mentioning C} \\
&\ldots \\
&\text{statements mentioning C} \\
&\text{endo C} \\
&\text{endpardo A,B} \\
\end{aligned}
\]

provided analysis has shown that the overall loop semantics will not be affected.

• Introduce local accumulator Consider the following variation of the SIAL example shown earlier.

\[
\text{pardo A,B,I,J} \\
\text{do C} \\
\begin{aligned}
&\text{do D} \\
&\text{get T}(C,D,I,J) \\
&\text{compute integrals V}(A,B,C,D) \\
&\text{temp}(A,B,I,J) = V(A,B,C,D) \ast T(C,D,I,J) \\
&\text{put R}(A,B,I,J) += \text{temp}(A,B,I,J) \\
&\text{endo D} \\
&\text{endo C} \\
&\text{endpardo A,B,I,J}
\end{aligned}
\]

Although this is a more straightforward coding of the original mathematical expression, the inner loop ac-
cesses the distributed array $R$. This potentially requires interprocessor communication on every iteration. To refactor this, the programmer would select $R$ and the refactoring function would introduce the local temp array $\text{tempsum}$ and transform the code to obtain the previous version.

### 3.2 Structural refactoring

**Extract procedure** In this refactoring, the programmer selects a segment of code and chooses the "extract procedure" refactoring command from the refactoring menu. After determining that the selected code is well-formed (it cannot contain part of a do loop, for example) the user is prompted for the procedure name. If the name is legal, then the selected code is replaced with a call statement at that point, and a procedure declaration for a procedure with the given name and selected statements as a body is inserted in the list of program declarations. Since SIAL procedures are actually macros and do not allow parameters or return values, this refactoring is quite simple.

**Inline procedure** In this refactoring, the programmer selects a call statement and chooses the "inline procedure" refactoring command. The call statement is replaced with the body of the matching procedure. The programmer indicates whether to inline all instances of the procedure call, and whether to delete the procedure declaration. The only potential complication would be attempting to inline a procedure containing a return statement at some location other than the last line of the procedure body. Since an examination of a large number of SIAL programs did not reveal any occurrences of this, rather than implementing a complicated transformation, the refactoring implementation simply checks for return statements and refuses to perform the refactoring if it finds one.

### 3.3 Data access

SIAL recognizes several kinds of arrays with different storage characteristics. 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. Distributed arrays are distributed over the memory of all tasks. Served arrays are used for very large arrays that will be stored on disk. They are accessed by the SIP in the most efficient way for the particular system. Data access refactorings modify the storage type of arrays.

**Change distributed array to served** This modification may be needed when the problem size becomes too big for a certain array to be stored in distributed memory of the target machine. In addition to changing the type declaration of the array, this involves changing the statements that create and access the array, for example, in SIAL, `request` and `prepare` statements are used to obtain and store blocks of a served array, while `get` and `put` are used for distributed arrays. Although this refactoring can allow larger problems to be handled, the performance may be less than desired. Thus this refactoring will often be followed by the Cache served array blocks refactoring, described next.

**Cache served array blocks** Experience has shown that frequent direct access of served arrays can be slow and performance can be improved by caching served data in a distributed array. Since the entire served array is presumably too large to fit in distributed memory, only a subset of its blocks can be cached at a time. Since SIAL requires index bounds to be statically declared, this refactoring is more tedious than one would expect, and has led to recent proposals to enhance the SIAL syntax to handle this situation more conveniently. Refactoring support will still be a significant convenience to the programmer.

### 4. Conclusion

Until now, SIAL programs have been successfully implemented, but tuned and refactored manually without tool support. We expect that appropriate refactoring support will enhance productivity of experts and novice alike. The experts will be able to do what they would have done manually, but faster and more accurately. The existence of certain refactorings can also serve as a guide to novices about the possibilities. The IMP IDE metatooling platform that we are using for this effort makes it easy to do things like provide hover help.

SIAL is not the only attempt to introduce a domain specific language for electronic structure computations. The Tensor Contraction Engine [BAB+05] provides a domain specific language that serves as input into a translator that generates FORTRAN code utilizing the Global Arrays toolkit to manage distributed arrays. The TCE aggressively attempts to optimize the generated code. However, the complexity of the optimization
problem where computation, communication, and storage requirements must all be balanced means that the results may not be as good as can be obtained by an experienced parallel programmer in the domain. The optimized code is typically too complicated to be suitable as a starting point for manual tuning.

If refactoring tools can incorporate and expose, in a useful way, some of the analyses done in an optimizing compiler while still leaving the programmer in charge, this has potential to hit the sweet spot between completely manual and completely automatic code optimization. This will increase programmer productivity and enable scientific discovery.

References


