SIAL Programmer Guide

ACES III Documentation

Quantum Theory Project
University of Florida
Gainesville, FL 32605

Contributors to the software:
R. J. Bartlett, R. Bhoj, E. Deumens, N. Flocke, T. Hughes, N. Jindal,
V. F. Lotrich, D. I. Lyakh A. Perera, J. M. Ponton, B. A. Sanders, T. Watson

Software version: 3.0.5 Oct 2010
Document version: 3.0.5 E Feb 2011
Document formatted:

May 21, 2011

Contents

1 Overview

Sections still to be written

- 7 Document ACES III SIAL programs by Tom

This guide explains how to write programs for the Super Instruction Processor. The full specification of the Super Instruction Assembly Language or SIAL, pronounced "sail", is given in Sect. 3. Example SIAL programs can be found in Sect. 8.

This user guide is one of a set of two. This document focuses on the SIAL programmer as the user of the super instruction architecture (SIA) parallel programming environment. The companion ACES III User Guide focuses on the user of the ACES III application that is written in SIAL to use the SIA environment for solving compute intensive problems in electronic structure theory using Coupled Cluster methods.

The language was created to easily write complex algorithms for the super instruction processor or SIP. Rather than an interpreter that executes every command when entered, this processor is more like a hardware processor, such as a modern micro chip. The instructions are called super instructions because they operate not on individual numbers, but on data blocks containing many thousands of numbers.

The language is more like an assembly language and is called the super instruction assembly language (SIAL). The SIP calls the compiler to read the SIAL program in its entirety and then the processor executes the binary code produced. A stand alone compiler is also available and it writes a .sio file that can be read and executed by the SIP.

The super instruction architecture (SIA) provides an interface for execution optimization of very specialized code that requires a lot of processing and a lot of data communication. All operations are performed in relatively large blocks, compared to the basic unit of a 64 bit computer word, and the operations are performed asynchronously allowing for multiple instructions being executed in parallel in each of the tasks in the SIP.

The goal of the SIA is to allow efficient parallel processing where latency plays less of a role and enough work is being given to all components so that there is sufficient time to hide the latency of any operation. However, it is still necessary to ensure that the bandwidth of all operations matches so that a steady state exists. This is where some, automated, tuning of problem to hardware comes in. The problem will be divided up into such pieces that the given hardware can sustain a steady state where all latency is hidden.

This document consists of three major parts: First the general structure of writing software using the SIA is explained in Sect. 2. Then the definition of the SIAL language is presented in Sect. 3. The execution environment is explained in Sect. 4. Next the software development environment is described in Sect. 5. Finally some reference material is collected including a list of special instructions (Sect. 6), a list of all domain specific SIAL programs written to provide the functionality of ACES III (Sect. 7, the full listing of a few example SIAL programs and an example super instruction (Sect. 8), and the format of the SIAL

object file produced by the SIAL compiler (Sect. 9).

PROPOSED

1.1 Preparing for a new version of SIAL, SIP, and ACES III *

This document describes the version 3.0.5 of ACES III which has SIAL and SIP built into it. Part of the function of the document is to prepare for and describe the new version where SIAL and SIP become stand alone parallel programming environment and where the domain specific capabilities of ACES III are more identifiably separate from the general framework.

Proposed changes and extensions

- 2.2 SIAL development environment by Erik and Beverly
- 2.2.1 SIAL compiler by Beverly
- 2.2.4 Domain specification file by Beverly
- 2.3 SIP as a python extension by Rohit
- 3.3.3 PERSISTENT qualifier by X
- 3.3.7 Support for high-rank arrays by X
- 3.4.2 PARDO with processor-groups by X
- 3.4.3 PARDO with grouping by X
- 3.5.1 Parallel library call by X
- 3.5.2 Move barriers from special super instructions in to the language by X
- 3.5.4 Argument list for special super instructions by X
- 3.5.5 Move compute_integrals to a special super instruction by X
- 3.6 PARALLEL SECTIONS construct by X
- 4.1.4 Support for GPUs by Nakul Jindal
- 4.3.5 Fault tolerance by X
- 4.3.6 ScaLAPACK interoperability by X

2 Super instruction programming environment

Writing parallel programs in the super instruction architecture involves three components:

1. a program written in the super instruction assembly language (SIAL) that determines the execution and data flow,
2. a library of super instructions, which are subroutines with compute kernels, needed by that SIAL program, and
3. a runtime environment to execute the SIAL program called the super instruction processor (SIP).

2.1 Programming guidelines

Programs must be viewed as having only one global scope. There are procedures, but these should be viewed as tools to organize code and logic, there are no local variables. All variables must be declared at the beginning of the program, even index loops. This is in agreement with the idea that every data element is a heavy object: It is a block of floating point numbers of considerable size, and any operation on it is expensive and must be considered with care. Therefore, creating temporary copies for passing them as arguments is too expensive.

Temporary variables, blocks, do exist and should be used as much as possible. They are allocated when first needed and the SIP will regularly check whether they are still needed and if not, mark the blocks as available for use.

Further tools for programmers are the performance analysis tools discussed in Sect. 5.5 and the integrated development environment (IDE) module for Eclipse described in Sect. 5.1.

PROPOSED

2.2 SIAL development environment *

Index of proposed changes with author 1.1
The following is an outline of what is needed for a development environment that would enable developers to use our runtime code to achieve scalable algorithms using the SIAL paradigm.

1. **Input data**
   Each SIAL program currently reads parameters from a ZMAT file and basis set data from a GENBAS file. Also, small amounts of data that may be used in other SIAL programs or even other application programs is read/written on a JOBARC file. Large data arrays that are to be passed from one SIAL program to another reside in blocked format on a BLOCKDATA file, and can be used to perform job restarts.

In a development environment, we envision an API that allows users to provide routines that can read parameter data in their own format and communicate those parameters to the SIP runtime code. Similarly, routines can be provided to read arrays of data, such as a grid, from disk files. This data can be communicated to the SIP, converted into blocked format, and stored in either distributed memory data structures or dumped off to the SIP I/O servers. Once this data has been fed into the SIAL environment, there must be a way for the SIP to understand how to link this data with the arrays defined in the SIAL code itself. A list of equivalences can be specified to do this, so that the data passed in a specific API call can be made equivalent to a specific SIAL array.

2. **Output data**
   We view output data as one of three types: (1) Data arrays that must be stored on disk for later processing, (2) small amounts of data that are computed in the job, and are needed in later processing steps (think of our TOTENEG and GRADIENT data), and (3) data printed by the job. Data of type (1) must be handled by user-provided routines through an API, similar to the input data. The user should have the ability to ask the SIP code for a particular slice of his data. The SIP routine will extract the data from its various blocks and pass it back to him. The user routine must be responsible for storing it in his own particular format. It would seem that data of type (2) must be extracted from the user's own data structures (common blocks, etc.), so it seems like a special instruction would be needed. Type (3) data could be printed via a special instruction as well.

3. **Special instructions**
   The way special instructions are handled now is that the routines are loaded into a table at compile time, and the table indices are compiled into the instructions. At runtime, the addresses of the instructions are loaded into the table as well, and when a special instruction is executed, its address is looked up in this table, and the routine is called with a standard set of arguments. The special instruction has access to all the SIP code's tables (array table, index table, etc.). Most of the special instruction routines consist of a standard preamble of decoding arrays out of the instruction table, looking up the current block of the array, finding its address in memory, etc.

Since most developers are not concerned with these details, we may want to consider refining this process, so that a developer is simply called with the various arrays used in the instruction, along with the indices and segments of those arrays and the data blocks themselves. This would greatly simplify writing a special instruction. Also, we can set up a "developer" directory in which these routines and the other developer-added software can be placed. Anything in this directory automatically gets linked in. An alternative to this is the Python-like scheme of dynamically loading these routines as shared objects.

4. **Indices and internal constants**
   We currently have a fairly small number of index types, which are preset and hard-wired, with a fairly rigid numerical ordering setting up the relationships between occupied, virtual, and AO and MO orbitals. Also, we use a number of pre-defined constants which are linked to our parameters, such as CC_ITER, CC_BEG, SCF_HIST, etc.

   Instead of having the indices hard-coded into the SIAL language, perhaps we can make it more adaptable by having an index definition file or interface. The compiler could get the definitions for its indices off of this file and use this to build SIAL programs. A similar idea could be used for the pre-defined constants and pre-defined static arrays. This will require some reworking of both the compiler and SIP code.

Once these elements are in place, we will have a reasonable application development platform. We will almost certainly run into other snags as people use this to add new algorithms. A good first test would be to restructure ACES III to fit within this framework.

2.2.1 SIAL Compiler

Index of proposed changes with author 1.1
Possible options are to generate a new compiler with the domain specific changes (DSC) or to provide a mechanism to specify the DSC in a header file that gets inserted in the SIAL program and processed by a compiler that remains unchanged. A combination is possible as well. Handling domain specific constants can possibly handled in a different way than a new set of indices. Note that SIAL supports three kinds of indices, segmented with variable segment-size, segmented with uniform segment-size and simple. The simple indices can be thought of as indices with a segment-size of one. Given this framework, different index types can be constructed by specifying the kind and the ranges can be specified with constants. Therefore it is not likely that support of domain specific indices requires special handling beyond the specification of constants.
It is important to realize that we need a mechanism for error checking that ensures that changes in the compiler are consistent with changes made in other components listed below.

2.2.2 SIO Object File

The format of the content of the object files created by the compiler is described in Sect. 9. We do not anticipate any need for the domain developer to make changes in this format, but we may need to make some changes to communicate changeable features in SIAL to the SIP.
Maybe a central repository or process must be defined to keep track of internal codes used for naming things like operation codes and index types.

2.2.3 SIP Runtime processor

The SIP goes through the following steps, some of which may need DSC while others seem to be unaffected.

1. Read input constants: An API is needed so that the domain developer can write a subroutine that SIP will call in which the domain input file is processed. In turn, the SIP needs to provide an API for the developer to call inside this routine to enable him/her to read and write values that correspond to the defined constants in the matching SIAL.

2. Initialization of blocks and stacks. This seems to need no change. Any custom action needed at this stage should be prepared in previous stages. For example, for indices with variable segment sizes, like the aoindex, an array with the length of each segment should have been prepared and should be available to the routines performing the initialization.

3. Perform a dry run pass through the SIAL code to estimate the use of memory and size all stacks. This, too, seems to need no change.

4. Read input data. With memory set up, further data, typically larger amounts than the set of constants read in the first reading, can now be read in. This requires a similar two-ended API.

5. Execution of the SIAL program by reading and processing the instruction table. This requires no change, but may require some error processing when instructions, predefined or special, are called in the SIAL program but do not exist or do not match routines known to SIP. The list of special instructions available to the customized SIAL program can be given in a header file. Then all instructions and anything special about them is available to SIP in the SIO-file. After one SIAL program completes execute the next one read from the input in step 1.

6. Decide whether the sequence of SIAL programs must be run again, e.g. as in a geometry optimization. It is possible that some data must be written to storage for the next sequence of executions or for future processing. Both these activities require a two-ended API as for reading constants and data.

2.2.4 Domain specification file

Index of proposed changes with author 1.1

The extensions are specified in a Domain Specification file (DSF) with a well-defined format that is easy to read but able to be mechanically processed. This file contains the name of the domain, an ID number, new index type names and the relationships between them, predefined variables including their default values, if any, and constraints on the values. An example is given below which should illustrate the concepts. This should really be done with an XML schema.

```
domain example:2000   //domain names and ids are centrally administered
                include aces:100
{   type
     {   index_type  aoindex2;   //type id 2000, the ids are generated
                                 //from the domain id
         index_type  moindex2 = 0;  //type id 2001, should default
                                    //values for the type be specified?
         index_type  moaindex2;  //type id 2002
         predef_index_type norbindex2;  // type id 2003
                                        //a predef_ type can only be
                                        //used for predefined variables
         scalar_type  velocity;   //type id 2004
         scalar_type  magnitude;  //type id 2005
     }
    compatibility
    {   norbindex2 < moindex2;  //a norbindex can be used where an
                                //moindex is expected
        norbindex2 < moaindex2; //a norbindex can be used where an
                                //moaindex is expected
        aoindex2 == moindex2;   //an aoindex can be used where an
                                //moindex is expected
                                // and vice versa
        velocity < scalar; // a velocity can be used whenever a scalar
                           //is expected
        magnitude < scalar; //a magnitude can be used whenever a
                            //scalar is expected
                            //but using a velocity when a magnitude is
                            //expected is an error
                            //default is t1 < t2
    }
    predefined
    {    norbindex norb2 ‘‘total number of atomic orbital segments’’;
                            //each predefined has a type, name,
                            // and a description
         int cc_beg2 = 2     ‘‘constant’’;  //optional default value
         moindex2 nocc2     ‘‘number of occupied molecular orbital
                                                  segments (no spin)’’;
         moindex bvirt      ‘‘begin of virtual orbital segment range
                                                        (no spin)’’;
                            //this one came from the included definition
         moaindex naocc   ‘‘number of occupied molecular orbital
                                                    segments (alpha)’’;
    }
```

```
    constraints
    {  * <= norb;  //  *  is a wild card
       0 <= *;
       1 < nocc;
       1 < nocc;
       bvirt <= nocc;
   }
}
```

The DSF can be processed (manually, or eventually automatically) to generate a variety of additional files that are used in the SIAL compiler, the SIP, and the IDE. Since all will be generated from a single DSF, no inconsistencies will be generated.
To ensure consistency between all of the generated parts at execution time, each DSF will be generate a key obtained by applying a hash function to its contents. This key will be included in each file generated and checked for consistency at run (or link) time. If the DSF changes, the key will also change, and thus we avoid inconsistencies caused by using different components that are generated from different versions of the DSF.
From the DSF, something along the following lines should be generated:

1. Header files analogous to keywdcount.h which contain the types and their associated constant values that can be used throughout the SIA.

2. A class (possibly a replacement for QCArrayClass) that encodes the type compatibility information. Existing parts or the compiler that do type checking need to be modified to use this instead of hard coded conditions.

3. A class that encodes the given constraints on the the predefined variables. Existing parts of the compiler that check array bounds need to be modified to use this instead of hard coded conditions.

4. syntax.cpp currently contains IsPreConstant and IsDeclaration methods. These should be provided by the previously mentioned classes

5. Files needed by the IDE

It would be easy to create a simple IDE for DSF files.
The values for predefined constants are currently obtained from a ZMAT or JOBARC file. Instead of providing an API for this, the users should provide something that looks like a generalized ZMAT file containing (name,value) pairs, and including the ability to read small arrays. Generalize framelib/params to read the new files (while maintaining compatibility). It should also be able to check that all predefined constants either have a default value or have been given a value.

2.3 SIP as a Python extension *

Index of proposed changes with author 1.1

By enabling SIP to be imported into Python the power of SIAL programming can be made available to many other developers. The process to import SIP into Python can also be used to import it into NWChem and MPQC, thus making the capabilities in coupled cluster theory implemented in SIAL programs listed in Sect. 7 directly available to these large user communities.
To show the functionality provided by creating an environment like SIA as a dynamically loadable extension of another software package, we will describe in some detail an example ACES III execution from Python using the SIP as an extension of Python to run a set of ACES III SIAL programs. The details of the how to build SIP as an extension are not shown, but they are well-known and extensively documented in the Python community. Once the functionality exists within Python, it is straightforward to port it to other environments such as NWChem and MPQC.
The user who wants to execute ACES III computations using the Python extension, will first start an interactive Python session from the command line. A batch session works the same way, except that the input for Python will come from a Python script file.

2.3.1 Functional view

At the Linux interactive shell prompt, the user types the command to start a Python session:

```
mycomputer% python
>>>
```

and receive the basic Python prompt. To teach Python all the tricks SIP can do, the user uses the Python import command.

```
>>> import sip
```

This will cause the running python executable to dynamically load the shared object sip.so and the generic library libsip.so, which has the basic code and capabilities of SIP that is currently available in the executable xaces3.
Next the user indicates that she wants to use SIP in the quantum chemistry domain by configuring SIP. All commands provided by the SIP are now available as members of the loaded Python module called sip.

```
>>> sip.configure('quantchem.def')
```

This command instructs the SIP extension of Python to load the domain specific shared object libaces3.so, which contains the code required to support orbitals, compute integrals, read ZMAT files, write JOBARC files, etc. Next the user needs to create a data structure that will hold all the information for a computation, sort of like a JOBARC file but resident in memory. We call it rdx in the example because we intend to do a calculation on the molecule RDX,

```
>>> rdx = sip.data()
```

Then the user loads the information from the ZMAT file that we created for RDX in the usual ACES III format into the Python/SIP data structure called rdx and allow SIP to do some appropriate initialization.

```
>>> rdx.zmat('rdx.zmat')
```

Notice that the rdx data structure is an object that has a member function called "zmat" that knows how to process ACES III ZMAT files and loads the information from it into the data structure. The domain specific method "zmat" is provided by the libaces3.so shared object loaded by the "configure" method above. The "configure" method is a generic SIP method and is provided by libsip.so.
The next step prepares SIP to run a SIAL program. A SIAL program consists of at least two source files:

1. a source file written in SIAL, for example scf.sial, contains the over all logic of the algorithm, and

2. a source file written in Fortran, C, or C++, for example libscf.f, contains the source code for the special super instructions called from the SIAL program that are not provided by the generic SIP environment in libsip.so, not by the domain specific environment in libaces3.so.

In the SIAL program a new declaration makes the SIAL compiler aware of the location of the special super instructions as follows:

```
      USE 'libscf.f'
```

The user can compile the SIAL program with the generic SIP method "compile".

```
>>> sip.compile('scf.sial')
```

The compiler produces two files

1. scf.sio, the super instruction object file containing the code for the SIP, and

2. libscf.so, the shared object with the code of the special super instructions compiled for the particular hardware platform such as Intel processor possibly with nVidia GPU code attached to it as well.

It is not necessary for the user to recompile the SIAL programs before every run, we just show the compilation step in the example here to illustrate the flexibility provided by the SIP environment as an extension to Python.
With the data entered into the runtime data structure rdx, the SIP is almost ready to perform a calculation. Assume that the user wants to run the SCF SIAL program. To enable this, SIP must be prepared or initialized with the "init" method.

```
>>> rdx.init('scf.sio')
```

This causes SIP to dynamically load the shared object libscf.so (the name is contained in the .sio file) with the special super instructions and to verify that all required super instructions are now available. It also makes SIP go through the SIAL program in dry-run or memory-estimation mode. After the dry run, it allocates the block stacks and assigns all MPI tasks their role as worker or IO servers. Then the SCF SIAL program is executed.

```
>>> rdx.run()
```

The results from the calculation are now stored in the data structure rdx. The user can dump this information in the JOBARC format for processing by other tools or for compatibility with ACES III runs using the xaces3 executable instead of the Python extension.

```
>>> rdx.dump('rdx.jobarc')
```

The user can check the results from the rdx.jobarc file, or by using data members of rdx directly in Python commands. For example, to print the total energy, she types

```
>>> print rdx.toteng
```

To clean up after the SCF run, the user issues the "clear" command. Then the next SIAL program can be run, for example to perform the CCSD computation.

```
>>> rdx.clear()
>>> rdx.init('ccsd.sio')
>>> rdx.run()
>>> rdx.dump('rdx.jobarc')
```

The "clear" method deallocates the block stacks, which are sized for the SCF calculations and unloads the libscf.so shared object. The CCSD SIAL program consists of the SIAL source code file ccsd.sial and the special super instructions source file libccsd.f. The "init" method loads the libccsd.so shared object and allocates the block stacks with sizes suitable for the CCSD calculation as coded in the CCSD SIAL program.
All these commands and more can be used inside Python programs to compose very complex simulations. For example an molecular dynamics algorithm written in Python can be easily modified to perform molecular dynamics with CCSD gradients computed by ACES III this way.

2.3.2 Parallel execution

To have a parallel program, SIP must be started in all tasks. This can happen in multiple ways. A serial main program is called in a master task, for example the Python interactive shell. At the same time a standard executable like xaces3 is start in all other tasks and they all wait for commands from the SIP in the master task running as an extension of Python.

Another scenario is that all tasks start Python and immediately import the SIP extension, Then the worker tasks wait for commands from the master as in the previous scenario. A further enrichment can be contemplated: A large groups of tasks is started running Python. Depending on the control of a Python program one group of tasks all import SIP and start working on some set of SIAL programs. Another group of tasks also imports SIP but now starts to work on a different set of SIAL programs. The Python program, which can also access MPI with its own communicators, can ensure that the two groups of tasks exchange data as needed. This allows the user to build very complex simulations, leveraging the power of Python and SIP.
In all scenarios listed, Python can easily be substituted with NWChem or MPQC.

3 Language definition

3.1 Syntax

1. Input is free form.
2. The lines must be less than 256 characters.
3. There are no continuation lines
4. Keywords and variable names are case insensitive.
5. All text after the pound sign (#) is considered a comment and is ignored. Blank lines and lines with only comments are ignored.
6. The language may in the future include data layout directives.
7. Every line is meaningful by itself.
8. Every name can be up to 128 characters long and consists of alphanumeric characters and underscores, the first character must be alphabetic.
9. Reserved words cannot be used as names.

3.2 Domain specific predefined constants

Some domain specific variables are defined from other input sources, such as the JOBARC file. All these constants count in segments, not in individual orbitals. A general syntax to allow development of support for other domains is being developed. See Sect. 2.2 for a more extensive discussion.

1. norb: total number of atomic orbital segments equal to the total number of molecular orbital segments and therefore norb can also be used in declarations of "no spin", "alpha", "beta" MO indices.

2. scfenerg: SCF energy read in from JOBARC.
3. totenerg: Total energy read in from JOBARC.
4. damp: value of DAMPSCF from ZMAT.
5. scf_iter: value of SCF_MAXCYC from ZMAT.
6. scf_hist: value of SCF_EXPORDE from ZMAT.
7. scf_beg: value of SCF_EXPSTAR from ZMAT.
8. scf_conv: value of SCF_CONV from ZMAT.
9. cc_iter: value of CC_MAXCYC from ZMAT.
10. cc_hist: value of CC_MAXORDER from ZMAT.
11. cc_beg: Constant equal to 2, there is no ZMAT parameter corresponding to this constant.
12. cc_conv: value of CC_CONV from ZMAT.
13. natoms: the number of atoms from ZMAT.
14. itrips: Starting occupied orbital to process in the triples codes.
15. itripe: Ending occupied orbital to process in the triples codes.
16. ihess1: Beginning value for Hessian index i to process.
17. ihess2: Ending value for Hessian index i to process.
18. jhess1: Beginning value for Hessian index j to process.
19. jhess2: Ending value for Hessian index j to process.
20. subb: Beginning sub_index range used in triples codes.
21. sube: Ending sub_index range used in triples codes.
22. sip_sub_segsize: Segment size of subindex segments.
23. sip_sub_occ_segsize: Segment size for subindices of an occupied index.
24. sip_sub_virt_segsize: Segment size for subindices of a virtual index.
25. sip_sub_ao_segsize: Segment size for subindices of an atomic orbital index.

3.2.1 Index constants:

The constants of type "no spin molecular orbital", "alpha molecular orbital", "beta molecular orbital" cannot be compared, they are of different types that correspond to the declarations (defined below) MOINDEX, MOAINDEX, MOBINDEX, respectively.

1. nocc,naocc,nbocc: number of occupied molecular orbital segments (no spin, alpha, beta)
2. nvirt,navirt,nbvirt: number of unoccupied or virtual orbital segments (no spin, alpha, beta)
3. bocc,baocc,bbocc: begin of occupied orbital segment range (no spin, alpha, beta)
4. eocc,eaocc,ebocc: end of occupied orbital segment range (no spin, alpha, beta)
5. bvirt,bavirt,bbvirt: begin of virtual orbital segment range (no spin, alpha, beta)
6. evirt,eavirt,ebvirt: end of virtual orbital segment range (no spin, alpha, beta)

3.2.2 Ordering relations for index constants:

The following tests return true:

1. i $\leq$ norb
   where i is any of the predefined constants bocc, baocc, bbocc, eocc, eaocc, ebocc, bvirt, bavirt, bbvirt, evirt
2. i $\geq$ 1
   where i is any of the predefined constants bocc, baocc, bbocc, eocc, eaocc, ebocc, bvirt, bavirt, bbvirt, evirt, eavirt, ebvirt
3. i $\geq$ 0
   where i is nocc, naocc, nbocc, nvirt, navirt, nbvirt
4. The 6=4x3/2 relations between the 4 "no spin MO" constants are
   eocc $\geq$ bocc
   bvirt $>$ eocc
   bvirt $>$ bocc
   evirt $\geq$ bvirt
   evirt $>$ eocc
   evirt $>$ bocc
5. The 6=4x3/2 relations between the 4 "alpha MO" constants are
   eaocc $\geq$ baocc
   bavirt $>$ eaocc
   bavirt $>$ baocc
   eavirt $\geq$ bavirt
   eavirt $>$ eaocc

eavirt > baocc

6. The 6 relations between the 4 "beta MO" constants are
   ebocc ≥ bbocc
   bbvirt > ebocc
   bbvirt > bbocc
   ebvirt ≥ bbvirt
   ebvirt > ebocc
   ebvirt > bbocc

All tests that cannot be obtained from these by the logical operation of reversing the elements and the operator, are not defined.

3.2.3 Predefined arrays

A number of basic domain specific arrays are predefined. They are computed by one module and then used by many subsequent modules. They are written to the JOBARC file between the runs of different modules. Thus they are also available in restart runs that obtain some or all of their input from the JOBARC file.

These predefined arrays are deprecated and should be replaced by suitable local or distributed arrays.

1. static c(mu,p): Restricted spin orbital transformation matrix from the SCF, read in from JOBARC. p is moindex 1:norb and mu is aoindex 1:norb.

2. static ca(mu,pa): Alpha spin orbital transformation matrix from the SCF, read in from JOBARC. pa is moaindex 1:norb and mu is aoindex 1:norb.

3. static cb(mu,pb): Beta spin orbital transformation matrix from the SCF, read in from JOBARC. pb is mobindex 1:norb and mu is aoindex 1:norb.

4. static e(p): Restricted spin orbital energies from the SCF, read in from JOBARC. p is moindex 1:norb.

5. static ea(pa): Alpha spin orbital energies matrix from the SCF, read in from JOBARC. pa is moaindex 1:norb.

6. static eb(pb): Restricted spin orbital energies matrix from the SCF, read in from JOBARC. pb is mobindex 1:norb.

3.3 Declarations

1. aoindex mu=1,norb
   Define the AO block index mu with range 1 through norb. Note that these indices count blocks, not individual orbitals. Ranges must be defined using predefined constants and the number 1, all other values generate an assembly error.

2. moindex p=1,nocc
   defines the MO block index p with range 1 through nocc. Note that these indices count blocks, not individual orbitals. Ranges must be defined using predefined constants and the number 1, all other values generate an assembly error.

3. moaindex pa=1,naocc
   defines the MO alpha block index pa with range 1 through naocc. Note that these indices count blocks, not individual orbitals. Ranges must be defined using predefined constants and the number 1, all other values generate an assembly error.

4. mobindex pb=bbvirt,ebvirt
   defines the MO beta block index pb with range bbvirt through ebvirt. Note that these indices count blocks, not individual orbitals. Ranges must be defined using predefined constants and the number 1, all other values generate an assembly error.

5. index i=1,10
   defines a simple index i with range 1 through 10 to be used in DO loops e.g. for an iteration.

6. laindex l=1,23 defines an index l with range 1 through 23 that has no association with atomic or molecular orbitals, but can be used to declare a dimension of an array. With this type of index, arrays can be created that have a mixture of dimension indices: some dimensions can be specified with the range of AOINDEX or MOINDEX and other dimensions with LAINDEX. The convention is that the dimensions with type LAINDEX must come after all dimensions with type AOINDEX or MOINDEX.

7. sub_index sub = subb, sube
   defines a subrange of data which ranges over the pre-defined constants subb to sube. The values subb and sube are obtained via user input at run-time.

8. scalar fac
   defines the scalar variable fac of type real (integers are treated as real). This value is local to each task.

9. static fock(mu,p)
   defines an array stored locally in the task allocated with a separate malloc. All predefined arrays are of this kind.

10. temp v1(p,mu,lambda,sigma)
    defines an array block with one MO and three AO indices that only exists locally in the form of a single block allocated on the block stack.

11. local aa(mu,p)
    defines an array stored locally in the task and allocated on the block stack.

12. distributed v4(p,q,r,s)
    defines an array distributed over many tasks and allocated on the block stack. The way it is distributed is determined outside the SIAL.

13. served v(mu,nu,lambda,sigma)
    defines the array v with four AO indices, which must have been defined before and specifies that the array is in disk-backed storage. Access to the array will be through a dedicated set of tasks called IO servers. The servers hold blocks of the array and write them to a file system on disk. The IO servers have a mechanism to manage their local memory to optimize the flow of data blocks to and from the served arrays. Blocks will be delivered by an IO server on request. The association of blocks with IO servers is fixed during an execution run but is determined by the SIP, not be the SIAL program. Because served arrays can overflow to disk, they can be larger than distributed arrays.

3.3.1 Multi-segment indices

In the domain of electronic structure there are indices that have subranges that are treated differently in many computational expressions. For example, an

```
moindex p=1,norb
```

is divided into two subranges of the same type

```
moindex i=bocc,eocc
moindex a=bvirt,evirt
```

called the "occupied orbitals" and the "virtual orbitals". In a typical problem the predefiend constants are consecutive and bocc=1, eocc=nocc, bvirt=nocc+1, evirt=norb. These index types are the same type, but for optimal performance it may be advantageous to have different segment sizes associated with the indices that run over the subranges. This is particularly relevant because the higher level mathematical operations to be performed often imply slightly different operations on elements of arrays depending on which subrange or subranges the indices of elements belong to. Table 1 gives an example of a molecular orbital (MO) index and the occupied and virtaul subranges with their own segment sizes. The result is that any loop over the moindex p will use segment size of 10 for the first four values of teh (super)index, covering orbitals 1 through 40, and it will use index 20 for the (super)index values 5 through 10, covering orbitals 41 through 140.

moindex	p					
moindex	i			a		
index value	1	...	4	5	...	10
segment size	10	...	10	20	...	20
orbital number	1	...	40	41	...	140

Table 1: An moindex p and the moindices i and a covering the occupied and virtual subranges with their respective segments, which can have different sizes.

In principle, one could define separate arrays so that segment sizes for these arrays are the same throughout the range of all its indices. However, this is not natural for the mathematical expression and definition of algorithms in this domain. It is likely that other domains have similar needs.

It is the responsibility of the designer of the domain specific definitions and the programmer of the domain specific initialization code that determines the predefiend constants to ensure that a consistent set of subranges results.

Note that the SIAL programmer cannot define new multi-segment indices. They are defined at the time the domain specific features are set up. Once they are defined in a domain, the runtime system will provide an environment in which indices can be used without ever running into any problems.

3.3.2 Scoping rules

All indices and arrays have global scope. However, the arrays do not always have blocks of data associated with them.

1. temp arrays have a block upon the first assignment until the closest enclosed code block ends with ENDDO or ENDPARDO. Note that this does not hold for the IF code block.

2. local arrays have blocks after ALLOCATE until DEALLOCATE.

3. Distributed arrays have blocks after CREATE until DELETE.

4. Served arrays have some blocks after they are prepared until all the blocks in the array are removed with DESTROY; the arrays still exists and can be used again for new data.

SIAL has a single global name space. All variables must be declared at the beginning of the program.

A scope is the text of a loop between a PARDO or DO statement and the corresponding ENDPARDO or ENDDO, respectively. Scopes may be nested, although a PARDO loop may not be nested inside another PARDO loop. Value bound in a scope are also bound in any inner scope. In contrast to many programming languages, procedures do not form a scope; they are actually macros and can be best understood as being inserted where they appear (with a return statement interpreted as a jump to the instruction following the procedure.)

SIAL programs are divided into sections by barriers, respectively. No barriers are allowed inside PARDO statements. All branches of an IF statement must contain the same set of barriers (server_barrier or sip_barrier) in the same order.

A legal SIAL program must be correctly synchronized. The semantics of SIAL programs that are not correctly synchronized is undefined.

A correctly synchronized program has a sip_barrier or server_barrier between conflicting accesses to distributed and served arrays, respectively. Conflicting operations are two operations in the same section accessing the same array as indicated in Table 2.

In Tables 3, 4, 5 value bindings are determined with respect to the text of the program and the program semantics. For example, a block of an initialized distributed array is bound to a value by a GET statement, even though in the implementation, there may be a delay. This is because, after the GET, another statement may access the array block and be sure it will see a valid value. The fact that this might actually involve waiting is irrelevant to these semantics.

	put	get	accumulate (+=)	prepare	request
put	X	X	X	N/A	N/A
get	X		X	N/A	N/A
accumulate (+=)	X	X		X	X
prepare	N/A	N/A	X	X	X
request	N/A	N/A	X	X	

Table 2: Conflicting operations on arrays must be synchronized with sip_barrier or server_barrier.

type	bound to a value (undefined)	unbound	storage allocated	storage deallocated	location
predefined constants	prior to runtime	N/A	static	N/A	replicated
static array element	assignment	N/A	static	N/A	replicated
index	header of DO or PARDO loop	end of loop	static	N/A	replicated
block of temp array	first assignment in a scope where undefined	end of defining scope	Implicit when bound, obtained from block stack	Implicit at end of defining scope	local
block of local array	when allocated; is Initialized to zero	when deallocated	allocate statement, obtained from block stack	deallocate statement	local

Table 3: State changes of variables (1)

PROPOSED

3.3.3 PERSISTENT qualifier *

Index of proposed changes with author 1.1

SIAL programs are executed as single units. All communication of data happens through blocks_to_list files written at the end of a run by one program and read in at the beginning by the next SIAL program.

We can modify the architecture of SIAL programs by renaming what is now called a SIAL "program" to a new thing called a SIAL "overlay" and allow multiple overlays to be part of a larger unit that will be then a SIAL program. Obviously it will be possible

type	bound to a value (undefined)	unbound	storage allocated	storage deallocated	location
block of distributed array	Bound to current value by get. Binding to current value may occur immediately if the primary or a cached copy is local, or after communication with the worker holding the primary block if not.	End of scope where GET performed.	When a CREATE statement is executed, each block of the array is allocated on the block stack of one of the participating workers. This is considered the primary copy of the block. Each block is initialized to 0s. The SIP determines the data distribution (i.e. which worker has which block). A local copy of the block is cached in the block stack of the worker after a GET.	DELETE deallocates all blocks of the primary copy. Unbound cached copies may be removed from a worker's block stack when the block is needed for reuse, when a sip_barrier is performed, or when the array deallocated.	A primary copy with cached local copies.

Table 4: State changes of variables (2)

type	bound to a value (undefined)	unbound	storage allocated	storage deallocated	location
block of served array	No default initialization, only defined after a prepare has been executed by some worker. Bound to current value on request or PREQUEST	End of scope where REQUEST/ PREQUEST performed, or until server_barrier, whichever comes first.	Each primary block is allocated at the IO server when the first prepare is performed. A local copy of the block is cached in the block stack of the worker after a REQUEST or PREQUEST.	Cached copies are removed from a worker's block stack when the block is needed for reuse, or when a server_barrier is performed.	The primary copy of a block managed by an IO server with cached local copies at workers

Table 5: State changes of variables (3)

to have SIAL program that contains only a single SIAL overlay to reproduce what we have now.
The functionality added by this hierarchical structure would be that arrays can be declared in the SIAL program as PERSISTENT, or something similar, with the meaning that they will persist across multiple overlays and can then be IMPORTED in overlays that need the data as needed. The initial implementation would use the existing mechanism of writing out blocks_to_list files to preserve arrays. The structure of program and overlay would provide a framework for more a general implementation.
Exit points of overlays are also natural points to call for check-points. The infrastructure to define and communicate data between overlays will be helpful in creating more powerful check-pointing capabilities.
For example, the orbital coefficients are now declared as a predefined array and are implemented as a replicated static array. They could become a persistent array created in the SCF overlay and used in all later overlays. This would allow it to become a distributed array to support much larger molecules.
The construct of an overlay is also relevant to define a richer environment of hierarchical parallelism. It is possible to express dependencies among overlays inside the SIAL program. For example the dependency of a CCSD gradient calculation may be

```
SIAL ccsd_grad
  PERSISTENT DISTRIBUTED c
  PERSISTENT DISTRIBUTED tijab
  ...
```

```
  OVERLAY scf
  OVERLAY ccsd NEEDS scf
  OVERLAY lambda NEEDS ccsd
  OVERLAY onegrad NEEDS lambda
  OVERLAY twograd NEEDS lambda
END SIAL ccsd_grad
```

This will allow SIP to execute onegrad and twograd in parallel on different groups of workers when there are sufficient workers in the run. Each overlay will IMPORT its version of the T-amplitudes as a distributed array, which will then be replicated twice in the whole system, once in each group of workers executing the overlay.
We can call such overlays that can run on a subgroup of all workers a partial overlay.
The infrastructure of a partial overlay is also useful when running on a large number of processors, e.g. 60,000, for a complex calculation like a CCSD(T) geometry optimization and some of the overlays, e.g. SCF, for not perform well at this scale or even anti-scale. Then the SIP can run the SCF overlay as a partial overlay on a subgroup of the workers. All workers are needed for some of the overlays and splitting the job in multiple jobs with different numbers of processors is impractical.

3.3.4 Example formula using high-rank arrays (Victor)

In order to test our ability to use high-rank arrays effectively we propose that as an initial set of expressions to evaluate we consider the following, which consists of evaluating a 10 dimensional array(non iteratively) from smaller arrays and then contracting this with smaller arrays to form a scalar.

$$E = [[[[t^{abcde}_{ijklm} V^{lm}_{l_1 e}] V^{l_1 k}_{d k_1}] V^{k_1 j}_{c j_1}] V^{i j_1}_{ab}] \tag{1}$$

where

$$t^{abcde}_{ijklm} = [[[t^{ab}_{i j_1} V^{j_1 c}_{j k_1}] V^{k_1 d}_{k l_1}] V^{l_1 e}_{lm}] \tag{2}$$

In the above expressions the arrays t^{ab}_{ij} and V are known and will be determined independently from the above. Evaluation of E using minimal cost and storage requirements is the goal. To this end t^{abcde}_{ijklm} should be evaluated from Eq. (2) and then the energy expression Eq. (1) evaluated. Indices which are repeated over are assumed to be summed over.

In Eq. (1) and Eq. (2) the square brackets indicate how the calculation should be preformed. (How intermediates should be defined) Therefore in order to determine t^{abcde}_{ijklm} the following sequence of calculations(and intermediates) should be performed.

$$X^{abc}_{ijk_1} = t^{ab}_{i j_1} V^{j_1 c}_{j k_1} \tag{3}$$

$$X^{abcd}_{ijkl_1} = X^{abc}_{ijk_1} V^{k_1 d}_{k l_1} \tag{4}$$

$$t^{abcde}_{ijklm} = X^{abcd}_{ijkl_1} V^{l_1 e}_{lm} \tag{5}$$

The energy expression should be pre-formed using the following sequence.

$$Y^{abcd}_{ijkl_1} = t^{abcde}_{ijklm} V^{lm}_{l_1 e} \tag{6}$$

$$Y^{abc}_{ijk_1} = Y^{abcd}_{ijkl_1} V^{l_1 k}_{dk_1} \tag{7}$$

$$Y^{ab}_{ij_l} = Y^{abc}_{ijk_1} V^{k_1 j}_{cj_1} \tag{8}$$

$$E = Y^{ab}_{ij_1} V^{ij_1}_{ab} \tag{9}$$

This is not an exhaustive test but should serve as an adequate initial set of expressions to evaluate in order to determine if we have any reasonable way to handle multidimensional arrays.

3.3.5 Example formula using high-rank arrays (Dmitry)

To implement multirefrence CC methods, it is necessary to code expressions that involve very rank, up to rank 16 has been seen. Fortunately there are some conditions that limit the size of the arrays involved.

The conventions to indicate the type of indices are as follows:

- J - open active hole;
- j - open inactive hole;
- B - open active particle;
- b - open inactive particle;
- l - free general hole;
- K - free active hole;
- k - free inactive hole;
- d - free general particle;
- C - free active particle;
- c - free inactive particle.

An idex is called open when it has a fixed value, and free when it is summed or contracted over. The Total range of molecular orbitals is NORB and is divided into four regions shown in Table 6 General hole/particle index range is split into the direct sum of inactive and active subranges. The active subrange is small. The inactive subrange is a compliment to the active subrange.

All the formulae below correspond to the follwoing operator product:

d b, c virtual, inactive, large (also called particles)
d B, C virtual, active, very small (also called particles)
l J, K occupied, active, very small (also called holes)
l j, k occupied, inactive, small (also called holes)

Table 6: The types of orbitals in expressions that need high-rank arrays.

`Z+=L*H*S`

where

- H is the Hamiltonian operator;
- S is a 4-fold excited amplitude;
- L is a 8-fold excited Lambda coefficient;
- Z is a residual tensor corresponding to 6-fold excited Lambda coefficient.

Diagram nr 1

$$Z^{j_1 J_1 J_2 J_3 J_4 J_5}_{b_1 b_2 B_1 B_2 B_3 B_4} += L^{J_1 J_2 J_3 J_4 J_5 k_1 k_2 K_1}_{B_1 B_2 B_3 B_4 d_1 d_2 C_1 C_2} * H^{j_1 K_2}_{b_1 b_2} * S^{d_1 d_2 C_1 C_2}_{k_1 k_2 K_1 K_2} \tag{10}$$

Diagram nr 11

$$Z^{j_1 J_1 J_2 J_3 J_4 J_5}_{b_1 B_1 B_2 B_3 B_4 B_5} += L^{j_1 J_1 J_2 J_3 J_4 l_1 K_1 K_2}_{B_1 B_2 B_3 B_4 d_1 d_2 C_1 C_2} * H^{J_5 l_2}_{b_1 B_5} * S^{d_1 d_2 C_1 C_2}_{l_1 K_1 K_2 l_2} \tag{11}$$

Diagram nr 15

$$Z^{j_1 j_2 J_1 J_2 J_3 J_4}_{B_1 B_2 B_3 B_4 B_5 B_6} += L^{j_1 j_2 J_1 J_2 J_3 K_1 K_2 K_3}_{B_1 B_2 B_3 B_4 d_1 d_2 C_1 C_2} * H^{J_4 l_1}_{B_5 B_6} * S^{d_1 d_2 C_1 C_2}_{K_1 K_2 K_3 l_1} \tag{12}$$

Diagram nr 23

$$Z^{j_1J_1J_2J_3J_4J_5}_{b_1B_1B_2B_3B_4B_5} += L^{j_1J_1J_2J_3J_4l_1K_1K_2}_{b_1B_1B_2B_3d_1C_1C_2C_3} * H^{J_5l_2}_{B_4B_5} * S^{d_1C_1C_2C_3}_{l_1K_1K_2l_2} \tag{13}$$

Diagram nr 31

$$Z^{j_1J_1J_2J_3J_4J_5}_{b_1b_2B_1B_2B_3B_4} += L^{J_1J_2J_3J_4J_5l_1K_1K_2}_{B_1B_2B_3B_4d_1d_2C_1C_2} * H^{j_1l_2}_{b_1b_2} * S^{d_1d_2C_1C_2}_{l_1K_1K_2l_2} \tag{14}$$

Diagram nr 67

$$Z^{J_1J_2J_3J_4J_5J_6}_{b_1b_2B_1B_2B_3B_4} += L^{J_1J_2J_3J_4J_5J_6l_1K_1}_{b_1B_1B_2B_3d_1C_1C_2C_3} * H^{l_2K_2}_{b_2B_4} * S^{d_1C_1C_2C_3}_{l_1K_1l_2K_2} \tag{15}$$

Diagram nr 72

$$Z^{j_1J_1J_2J_3J_4J_5}_{b_1b_2B_1B_2B_3B_4} += L^{j_1J_1J_2J_3J_4J_5K_1K_2}_{b_1b_2B_1B_2C_1C_2C_3C_4} * H^{l_1l_2}_{B_3B_4} * S^{C_1C_2C_3C_4}_{K_1K_2l_1l_2} \tag{16}$$

Diagram nr 79

$$Z^{j_1j_2J_1J_2J_3J_4}_{b_1B_1B_2B_3B_4B_5} += L^{J_1J_2J_3J_4l_1l_2K_1K_2}_{B_1B_2B_3B_4B_5c_1c_2C_1} * H^{j_1j_2}_{b_1C_2} * S^{c_1c_2C_1C_2}_{l_1l_2K_1K_2} \tag{17}$$

Diagram nr 95

$$Z^{J_1J_2J_3J_4J_5J_6}_{b_1b_2B_1B_2B_3B_4} += L^{J_1J_2J_3J_4l_1l_2K_1K_2}_{b_1B_1B_2B_3B_4d_1C_1C_2} * H^{J_5J_6}_{b_2d_2} * S^{d_1C_1C_2d_2}_{l_1l_2K_1K_2} \tag{18}$$

Diagram nr 119

$$Z^{j_1j_2J_1J_2J_3J_4}_{b_1B_1B_2B_3B_4B_5} += L^{j_1j_2J_1J_2K_1K_2K_3K_4}_{B_1B_2B_3B_4B_5d_1C_1C_2} * H^{J_3J_4}_{b_1d_2} * S^{d_1C_1C_2d_2}_{K_1K_2K_3K_4} \tag{19}$$

Diagram nr 126

$$Z^{j_1j_2J_1J_2J_3J_4}_{B_1B_2B_3B_4B_5B_6} += L^{j_1J_1J_2J_3J_4l_1K_1K_2}_{B_1B_2B_3B_4B_5c_1c_2C_1} * H^{j_2l_2}_{B_6C_2} * S^{c_1c_2C_1C_2}_{l_1K_1K_2l_2} \tag{20}$$

Diagram nr 132

$$Z^{j_1J_1J_2J_3J_4J_5}_{b_1B_1B_2B_3B_4B_5} += L^{J_1J_2J_3J_4J_5k_1k_2K_1}_{b_1B_1B_2B_3B_4d_1C_1C_2} * H^{j_1K_2}_{B_5d_2} * S^{d_1C_1C_2d_2}_{k_1k_2K_1K_2} \tag{21}$$

Diagram nr 169

$$Z^{J_1J_2J_3J_4J_5J_6}_{B_1B_2B_3B_4B_5B_6} += L^{J_1J_2J_3J_4J_5l_1K_1K_2}_{B_1B_2B_3B_4B_5d_1C_1C_2} * H^{J_6l_2}_{B_6d_2} * S^{d_1C_1C_2d_2}_{l_1K_1K_2l_2} \tag{22}$$

Diagram nr 188

$$Z^{j_1J_1J_2J_3J_4J_5}_{B_1B_2B_3B_4B_5B_6} += L^{j_1J_1J_2J_3J_4J_5k_1K_1}_{B_1B_2B_3B_4B_5d_1C_1C_2} * H^{l_1K_2}_{B_6d_2} * S^{d_1C_1C_2d_2}_{k_1K_1l_1K_2} \tag{23}$$

Diagram nr 191

$$Z^{J_1J_2J_3J_4J_5J_6}_{b_1B_1B_2B_3B_4B_5} += L^{J_1J_2J_3J_4J_5J_6l_1K_1}_{B_1B_2B_3B_4B_5c_1c_2C_1} * H^{l_2K_2}_{b_1C_2} * S^{c_1c_2C_1C_2}_{l_1K_1l_2K_2} \tag{24}$$

Diagram nr 294

$$Z^{j_1J_1J_2J_3J_4J_5}_{b_1B_1B_2B_3B_4B_5} += L^{j_1J_1J_2J_3J_4J_5k_1K_1}_{b_1B_1B_2B_3B_4B_5c_1C_1} * H^{l_1K_2}_{d_1C_2} * S^{c_1C_1d_1C_2}_{k_1K_1l_1K_2} \tag{25}$$

Diagram nr 300

$$Z^{j_1 J_1 J_2 J_3 J_4 J_5}_{B_1 B_2 B_3 B_4 B_5 B_6} += L^{j_1 J_1 J_2 J_3 J_4 J_5 k_1 K_1}_{B_1 B_2 B_3 B_4 B_5 B_6 d_1 C_1} * H^{l_1 K_2}_{d_2 C_2} * S^{d_1 C_1 d_2 C_2}_{k_1 K_1 l_1 K_2} \tag{26}$$

Diagram nr 317

$$Z^{j_1 J_1 J_2 J_3 J_4 J_5}_{b_1 B_1 B_2 B_3 B_4 B_5} += L^{j_1 J_1 J_2 J_3 J_4 J_5 K_1 K_2}_{b_1 B_1 B_2 B_3 B_4 B_5 C_1 C_2} * H^{l_1 l_2}_{d_1 d_2} * S^{C_1 C_2 d_1 d_2}_{K_1 K_2 l_1 l_2} \tag{27}$$

It is clear from this sample of expressions that all possible contractions will occur and that there is no benefit in optimizing one combination over the others.

The S, L, and Z tensors obey the restrictions that they cannot contain more than two inactive or general holes and no more that two inactive or general particles. Since the active range of all indices is typically small, from 4 to 10, this limits all indices in these tensors to that small range for all indices except four, in the worst case of two inactive or general holes and two inactive or general particles. In Table 7 we show the sizes of data blocks for these arrays using some example values for segments, subsegments, and active ranges. For simplicity, we assume that the subsegment size is the same as the active range, which is reasonable for estimates since they are of the same order of magnitude and quite small. The storage reduction for an array with rank p from symmatric packing is estimated by reducing full storage need, given by N^p, by the factor 2^p. For some arrays, only partial symmetry can be used, then the formula is applied to the subset of indices.

	segment	subsegment	active range	rank full	rank active	block size (max/min)
H	28	-	-	4	0	5 MB / 0.6 MB
L	-	4	4	4	12	8 GB / 0.5 MB
S	-	4	4	4	4	0.5 MB / 2 KB
Z	-	4	4	3	9	34 MB / 32 KB

Table 7: Block sizes for high-rank arrays. We assume that the subsegment size and the active range are the same in this estimate, since they are of the same order of magnitude. An entry in the table as a dash means that that value does not contribute to determine the block size. The block size ranges from minimum, when all super indices are equal and the block can be packed with full symmetry, to maximum, when as many of the super indices are different as possible and the block can not be packed fully as a result.

3.3.6 Index schemes for packed storage of arrays

When high-rank arrays need to be stored the size of the needed storage grows quickly and it becomes increasingly important to ensure that the minimal amount of space needed is actually required. There is, however, a balance between compact storage and efficient access that mustbe taken into consideration. In this section, we will discuss the considrations and techniques that are commonly used to address this balance.

Every programming language that support multi-dimensional arrays uses the compiler to generate code to compute a combined index that progresses linearly through all elements of the array. For example, the array declared in Fortran 95 with

```
dimension a(10,10,10)
```

can be accessed in a loop as

```
sum = 0.d0
do k=1,10
  do j=1,10
    do i=1,10
      sum = sum + a(i,j,k)
    end do
  end do
end do
```

The compiler translates this into something that can be described as the literal translation of the following code in Fortran 95. The convention in Fortran is that the first index changes first or fastest. This is called "storage by column".

```
sum = 0.d0
do k=1,10
  do j=1,10
    do i=1,10
      sum = sum + a(i + (j-1)*10 * (k-1)*10*10)
    end do
  end do
end do
```

Thus the two sections of code will result in accesses of the memory locations of the array a in sequence starting with a(1), a(2), a(3), etc. Other languages like C/C++ work the same way with one important difference: The first index runs the slowest. This is called "storage by row". The same operation would then look like the following

```
sum = 0.;
for (i=0,k<10,k++) {
  for (j=0,k<10,k++) {
    for (k=0,k<10,k++) {
      sum += a[i*10*10 + j*10 * k]
    };
  };
};
```

Although the first code segment using a(i,j,k) looks more closely to the mathematical expression about the array, programmers have found that for large scale projects this can lead to inefficient computation, or wasteful use of memory for storage, or both. For example, the implied computation of the index i + (j-1)*10 * (k-1)*10*10 involves multiple operations that can become a problem if many arrays are being used. For this reason, optimizing compiler replace the code obove with something that looks like the literal translation of

```
sum = 0.d0
n = 0
do k=1,10
  do j=1,10
    do i=1,10
      n = n + 1
      sum = sum + a(n)
    end do
  end do
end do
```

Now the multiple multiplications and additions are replaced by a single increment of a counter in the inner loop. Other more sophistocated optimizations are in use, but this suffices for our purpose of showing that index manipulations must be given consideration in code with extreme high-performance requirements. Note that the optimization make the loop no longer parellelizable: Every iteration requires all previous iterations to be executed for the incremented counter n to point to the correct place. The code with the direct index computation can immediately be parallelized, because the index depends only on the values of i, j, and k and can be computed independently on many processors for different parts of nested loops. Of course, the sum computation needs some attention to give the correct result in a parallel execution.

The space optimization can be illustrated by considering the case where the array a has a symmetry. For example, consider a two-dimensional array, a matrix, that is symmetric a(i,j)=a(j,i). Then we can store 10*10=100 numbers of which only the 10 diagonal elements and the 10*9/2=45 elements below the diagonal are unique or we can store only the unique 45+10=55 elements. We can do this by declaring a as a single-dimension array, a vector, with length (10*(10+1))/2=55 and accessing it with the following index computations, using Fortran in the example,

```
double precision a[55]
sum = 0.d0
do j=1,10
  do i=1,j
    sum = sum + a(i+(j*(j+1))/2)
  end do
end do
```

Again optimizations to simplify index computations must be balanced with the need to allow parallelism. To make the code more readable, one often defines an intermediate index

```
double precision a[55]
sum = 0.d0
do j=1,10
  do i=1,j
    ij = i+(j*(j+1))/2
    sum = sum + a(ij)
```

```
  end do
end do
```

This same type of construction can be used recursively to store hig-rank arrays with multiple levels of symmetry compactly. For example, the 8-fold symmetry of the two-electron integrals is

$$\begin{aligned} v(i,j,k,l) &= v(j,i,k,l) = v(i,j,l,k) = v(j,i,l,k) \\ &= v(k,l,i,j) = v(l,k,i,j) = v(k,l,j,i) = v(l,k,j,i) \end{aligned} \tag{28}$$

This can be stored compactly by combining indices i and j into a symmetrically packed index ij as in the example of the symmetric matrix above. The same can be done for k and l. Then the two packed indices ij and kl can be packed again the same way into ijkl. Assuming a basic dimension of 10, the length of the 4-index integral array v becomes 55*(55+1)/2=1,540. The code then becomes

```
double precision v[1540]
sum = 0.d0
do l=1,10
  do k=1,l
    kl = k+(l*(l+1))/2
    do j=1,10
      do i=1,j
        ij = i+(j*(j+1))/2
        if (ij <= kl) then
          ijkl = ij + (kl*(kl+1))/2
          sum = sum + a(ij)
        end if
      end do
    end do
  end do
end do
```

Notice that we used a simple IF-statement to avoid including the cases ij ¿ kl. There are other ways to accomplish this but not a single best one that is simple and fast. In other words there is no simple bound on j that will ensure that we alsways have ij¡=kl.

Various optimizations are possible and different strategies can be used depending on the code in the body of the loops and on wether other arrays are involed that need similar intermdiate packed indices. Once two arrays are constructed with the packing then a full contraction can be done very efficiently.

$$\sum_{i<=j,k<=l,ij<=kl} v(i,j,k,l)a(i,j,k,l) \tag{29}$$

can be coded simply as

```
double precision v(1540), a(1540)
```

```
sum = 0.d0
do ijkl=1,1540
  sum = sum + v(ijkl)*a(ijkl)
end do
```

For antisymmetric matrices the index computation is

```
double precision a[45]
sum = 0.d0
do j=1,10
  do i=1,j-1
    ij = i+(j*(j-1))/2
    sum = sum + a(ij)
  end do
end do
```

Below is a summary of the basic index packing schemes.

```
ij_symm = i + (i*(i+1))/2
ij_asymm = i + (i*(i-1))/2
ij_col = i + (j-1)*N
ij_row = (i-1)*N + j
```

Because of the limitation in the Fortran language that arrays can have no more than 7 indices, some programs used the basic columnwise and rowwise index calculations to implement arrays of arbitrary and of dynamic rank.

High-rank arrays with various combinations of symmetric, antisymmatric, columnwise, and rows wise stored hierarchies of index pairs can be constructed. Managing the indices is not a simple task and requires attention from the program designed, because very poor performance can result and automatic optimization techniques in compilers usually cannot recognize and optimize hierarchies deeper than two.

PROPOSED

3.3.7 Support for high-rank arrays *

Index of proposed changes with author 1.1
Currently SIAL supports indices with up to 6 indices. This can be changed by a setting mx_array_index in ACESIII/include/maxdim.h.
For arrays defined with both simple indices and segmented indices, the simple indices must be at the end of the declaration. For example A(a,b,c,d,m,n) with a,b,c,d block indices and m,n simple indices is correct. As a reminder, segmented indices like AOINDEX and MOINDEX are indices that count segments; simple indices count like integers.
NOTE The original design for SIAL arrays included support for symmetric storage. This has not been implemented at this time, but will be very important to save space

and simplify expressions when high-rank arrays are being used. To implement this is well-understood and easy, so it will be not discussed in this section. We assume that that symmetric storage will be implemented as part of this effort. The symmetry applies both to the super indices labeling blocks and to the regular indices labeling the elements inside each block.

3.3.8 Proposal: Support arbitrary rank in SIAL

We propose the extend the SIAL support to have unlimited rank for arrays. The underlying structure of segmented indices and and block lists can easily occomodate providing the SIAL programmer with the luxury and arbitrary rank. The underlying super instructions can be written, possible with machine-gerenated code, using the standard techniques to overcome the limitation in the Fortran stanndard of 7 indices and will still produce high performance for tensor contractions at the block level.
The challenge is to find the proper data management approach to keep the performance of SIP as high as it is now. We analyze the problem next.
Consider the implications derived from the expressions in Sect. 3.3.5 in view of the requirements that any block-centered architecture such as SIA has to allow an efficient implementation.

1. All data blocks should have close to the same size, although smaller blocks are easy to support and will not impact performance if there ar not too many. The limit of small blocks is a single floating point number, which we know does not perform well and is the reason for considering SIA in the first place. Huge blocks on the other hand become unmanagable, clog up the communication system buffers and local RAM, and do not offer any performance advantages.

2. The current 4-index implementation works well for integral blocks H and T-amplitude blocks in the CCSD algorithm.

3. Mixing segmented indices with simple indices is suitable if the algorithm clearly distinguishes some indices as much more floating point intensive than others. Then the less intensive indices can be considered low-speed book-keeping indices and these can be treated as in scalar-orianted architectures (as opposed to the block-oriented SIA) without much performance loss.

4. Contractions of blocks should have sufficient number of indices to contract so that the underlying DGEMM can run optimally, contraction over a single index of blocks from arrays with high ranks is inefficient.

From the previous section, we see the following charactersitics of code that will be generated by the SIAL compiler, whatever language feature we use to specify the formulas in code.

1. High rank arrays will have a limited number of full-range indices. These can be defined with subsegments so that contractions with the Hamiltonian, defined with full segments, work optimally.

2. High rank arrays will have most indices of the type active, which have a small range that can easily be chosen to be equal to the size of a subsegment or just one or two subsegments.

3. Because of the high rank, the impact of symmetric packing on storage is very big. This results in blocks that a very different in size.

We are thus lead to making our storage scheme even more flexible and less regular, compared to Globale Arrays and ScaLAPACK block cyclic storage scheme, than we already have done with in the current implementation.

3.3.9 Proposal: Use compound indices in SIAL

In traditional Fortran and C implementations of electronic structure algorithms that require arbitrary large numbers of indices on tensors, such as high-order coupled cluster (CC) methods and configuration interaction (CI) up to full CI, have used compound indices. The basic strength of SIAL includes that operation on blocks and the use of super indices to express the algorithm. It is not immediately clear how the notion of super indices and blocks can be extended to compound indices.
We propose to follow the standard approach for defining multiple indices, similar to the method defined in Fortran, both for addressing the blocks as well as the numbers inside the blocks. Assume an array that is to be used as a four-index array and that each index is segmented in N sections of n elements. We then declare the block numbering and the indexing inside the blocks with the usual compound index expression

```
IJKL = I + (J-1)*N + (K-1)*N**2 + (L-1)*N**3
ijkl = i + (j-1)*n + (k-1)*n**2 + (l-1)*n**3
```

We can now address the blocks without the need for renumbering by various combinations of less-compound indices

```
(I,JKL) = (IJ,KL) = (I,J,K,L) = IJKL
```

Similarly, we can access each number inside a block by various compound-index combinations without the need for (expensive) rearrangement of data

```
(i,jkl) = (ij,kl) = (i,j,k,l) = ijkl
```

The examples above use the same segment size for all dimensions. This can be easily generalized. This generalization is important because without it the size of high-dimensional blocks will becomes too large for practical and high-performance work.
The compound-index scheme can be easily generalized to support forms of symmetric packing, which is very important for high-dimensional matrix objects like Coupled Cluster amplitudes. For example, the four indices of T_2 can be packed with the restriction that $i < j, k < l$, and $ij < kl$ to store every unique element only once.

```
ij = j*(j-1) + i
kl = l*(l-1) + k
ijkl = kl*(kl-1) + ij
     = (l*(l-1)+k)*(l*(l-1)+k-1) + j*(j-1) + i
```

3.4 Control statements

1. Program

   - sial myprog
     start of the SIAL program called myprog. With this control line the program can be embedded in any file and all text preceding this line is ignored by the assembler. The line must start with white space or the reserved word sial.
   - endsial myprog
     marks the end of a SIAL program. Everything in the file after that is ignored by the assembler.

2. Procedures

   - proc mywork
     start of a procedure called mywork. Procedures are only a tool to organize executable code, they are not to be compared to functions in C or subroutines in Fortran. They are like internal procedures in Fortran 90 and they operate in the one global scope of the SIAL program.
     - The proc end proc code block is inside the body of the sial end sial program definition.
     - Declarations are not allowed inside the proc body. The procedure can use temp arrays, but they must be declared in the scope of the main program.
     - All indices and arrays defined in the program are visible inside all procedures.
     - Procedures are like declarations and must be located after other declarations and before any executable statements.
   - endproc mywork
     end of the procedure called mywork. No other procedure can be defined inside it. All other control structures must be closed before this line, except end sial and that one may not be closed, i.e. the procedure must be inside the program definition.
   - return
     exits the running procedure and returns execution to the statement after the call to the procedure.

- call mywork
  calls the previously defined procedure mywork; at the end of the procedure or at execution of a return statement control returns to the line after the call to the procedure.

3. Distribution

- pardo mu,nu,lambda,sigma
  starts a distributed loop over the indices mu,nu,lambda,sigma. The work inside the loop is performed by every task only for those values of the listed block indices that have been assigned by the master to that task as controlled by the parallel environment directives.
- pardo mu,nu,lambda,sigma
  where condition
  starts a distributed loop over the indices mu,nu,lambda,sigma for all values of the indices that satisfy the logical condition after the "where" keyword. The work inside the loop is performed by every task only for those values of the listed block indices that have been assigned by the master to that task as controlled by the parallel environment directives.
- endpardo mu,nu,lambda,sigma
  ends the distributed loop with variables mu,nu,lambda,sigma; the variables must be specified. pardo structures cannot be nested, improperly nested loops generate an assembly error. Two consecutive distributed loops are allowed and can use the same index variables or different variables without error. Use of different names will not change the ranges assigned to each task.

4. Iteration

- do mu
  starts a loop over the index mu. The do statement is operationally equivalent to incrementing the loop index.
- enddo mu
  ends the loop with variable mu; the variable must be specified; improperly nested loops generate an assembly error. Two consecutive loops can use the same index variable without error.
- cycle mu
  makes control in the loop jump to the next iteration of the loop on the variable named on the cycle statement; a cycle statement without a variable name generates an assembly error.
- exit
  makes control in the loop jump to the statement after the end do of the closest loop enclosing the exit statement.

5. Conditions

   - if a<3
     starts an if-block with test on the scalar or index variables or expression on scalar or index variables. If the expression contains at least one scalar variable all computations in the expression are evaluated as C double or Fortran double precision, if the expression contains only index variables and integers, the expression is evaluated as C int or Fortran integer. Constants such as 3 or 3. are treated as integers and floats, respectively. The code inside the block is executed if the expression value is non-zero (true).
   - endif
     ends the if-block; improperly nested if-blocks generate an assembly error.
   - else
     starts the alternative code block in the if-block; improperly nested if-blocks generate an assembly error.

3.4.1 Subindices

SIAL handles large arrays by partitioning each dimension into segments. In SIAL source code, one might find a reference to X(i,j). Here, i and j are segment indices so X(i,j) actually refers to a block of data. For example, is the segment size in both dimensions is 10, then X(1,2) actually refers to the block X([1:10],[11:20]) if we view the array 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. 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 fails in situations where many dimensional, say 6-dimensional, arrays are generated. For example, A(a,b,c,k)*B(k,l,m,n) generates a 6 dimensional result, say C(a,b,c,l,m,n). If the segment size of the indices are chosen to be typical values for segments , then the block of C is so large that the computation becomes infeasible. 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.

The goal is to be able to conveniently partition the blocks of A and B, so that C could be computed in smaller blocks without changing the actual segment sizes of A and B.

To do this, we propose a set of new language features: subindex, do/pardo in, and slices and insertions.

subindex ii of i

- The superindex i must be a previously declared index with range [lower(i) , upper(i)] and segment size seg(i).
- ii is given a segment size seg(ii) specified using the same mechanism as seg(i) subject to the constraint that seg(i)%seg(ii)=0. In particular, a segment size of 1, corresponding to a simple index is allowed.

- The range of ii is [(lower(i)-1)*n+1,upper(i)*n] where n = seg(i)/seg(ii).
- The subindex inherits the type from its superindex.
- Subindices may be used to declare arrays. An array declared with a subindex will contain the same number of data items as an otherwise equivalent array declared with its superindex.

Example 1:

Suppose that baocc = 1, eaocc = 20, seg(i)=20, and seg(ii)=4. Then ii has range [1,100]. If baocc=2, then ii has range [6,100].

```
moaindex i = baocc, eaocc
subindex ii  of i
```

When fully formed, these arrays contain the same total number of data elements, but the blocks of Xii are the size of the blocks of Xi.

```
temp Xi(i)
temp Xii(ii)
```

do ii in i, pardo ii in i

- This is a loop over the values of ii that fall within the block specified by i, i.e. ii ranges from first = (i-1)*n+1 to first+seg(ii) where n = seg(i)/seg(ii).
- i must have a well defined value, which implies that this construct must be nested inside a do or pardo loop over i.
- i must be the superindex of ii
- It is also possible to use ii in a normal loop. This will iterate over the entire range of ii.

Example 2

```
do i
do ii in i
     Xii(ii)=0
enddo ii
endo i
```

is equivalent to

```
do ii
    Xii(ii)=0
enddo ii
```

Example 3

```
pardo i
do ii in i
    xii(ii)=0
enddo ii
endpardo i
```

This will perform a serial iteration over blocks 1,2,3,4,5 at one processor in parallel with a serial iteration over blocks 6,7,8,9,10 at another, in parallel with a serial iteration over blocks 11,12,13,14,15 at another, etc.

Example 4

```
do i
pardo ii in i
    Xii(ii)=0
endpardo ii
endo i
```

This construct would first initialize blocks 1,2,3,4,5 on five different processors in parallel, then initialize blocks 6,7,8,9,10 on five processors in parallel, etc.

Slices and insertions

```
Xii(a,ii) = Xi(a,ii)
Xi(a,ii) = Xii(a,ii)
```

where Xi was declared using indices a and i, Xii was declared with indices a and ii, ii is a subindex of i, and a is an arbitrary index.

- The right hand side of the first assignment refers to the portion of Xi determined by the subindex ii. Since Xi was declared with index i, but we are referring to it with index ii, this is slice. In this context, the subblock is copied into the smaller block on the left hand side.
- The second assignment takes the smaller block on the right side and inserts it into the appropriate location in the larger block on the right hand side. This is an insertion.
- In both cases, the compiler can easily determine that these are slicing or inserting assignments from the declarations.
- Design decision under consideration: Should use of a slice be restricted to the right side of assignments, or allowed anywhere where a block of compatible type is allowed? For example, would

  ```
  Yii(a,ii) = Xi(a,ii) + B(a,ii)
  ```

  be legal, or would we need to say

  ```
  Xii(a,ii)= Xi(a,ii)
  Yii(a,ii) = Xii(a,ii) + B(a,ii)
  ```

The former is more convenient for the programmer and may be potentially more efficient since it may reduce the amount of copying required if the super instruction implementing "+" is implemented in an appropriate way. If the super instruction implementation does not know about subindices, the compiler could generate the necessary copy instruction. The latter is more straightforward and will not require changes in any super instruction (except the slicing and inserting assignments).

Example 5

```
pardo a
do i
  get Xi(a,i)
  do ii in i
     Xii(a,ii) = Xi(a,ii)
     do something with Xii(a,ii)
  enddo ii
endo i
endpardo a
```

Example 6

In this example, we look at the CCSD_RHF_SV1 program that motivated these features. Instead of comparing with the actual working implementation using simple indices for two dimensions of a six dimensional array, we start with the straightforward solution that one have like to have written if it had been feasible. The example indicates how to change it to an implementation that is feasible using the new language features.

Straightforward, but infeasible version.

```
SIAL CCSD_RHF_SV1

      moaindex i= baocc, eaocc
      moaindex j= baocc, eaocc
      moaindex k= baocc, eaocc

      moaindex i1= baocc, eaocc
      moaindex j1= baocc, eaocc
      moaindex k1= baocc, eaocc

      moaindex a = bavirt, eavirt
      moaindex a1= bavirt, eavirt
      moaindex a2= bavirt, eavirt
      moaindex a3= bavirt, eavirt

      served VSaaai(a,a1,a2,i1)
      served TSaiai(a3,i1,a1,j1)

      temp taaaiii(a,a1,a2,k1,i1,j1)
```

```
# ------------------------------------------------------------------------
# BEGIN MAIN PROGRAM
# ------------------
#
#
# Read transformed integrals from lists
# -------------------------------------
#
      CALL READ_2EL #standard code for this proc omitted

      PARDO a, a1, a2

            IF a  <= a1
            IF a1 <= a2

# First do the expensive part involving 3-virtual integrals and
#    of order VVVVooo
# ------------------------------------------------------------------------

            DO a3
            DO k1

               REQUEST VSaaai(a,a3,a2,k1) a

               DO j1

                  IF j1 <= k1

                     DO i1

                        IF i1 <= j1

                           REQUEST  TSaiai(a3,i1,a1,j1) a3


       #this is where we have a problem, blocks of t1aaaiii are too big.
                           taaaiii(a,a1,a2,k1,i1,j1)=
                                 VSaaai(a,a3,a2,k1)*TSaiai(a3,i1,a1,j1)

                        ENDIF
                     ENDDO i1
                  ENDIF
               ENDDO j1
```

```
            ENDDO k1

            ENDDO a3
#
# ------------------------------------------------------------------------
#
            ENDIF
            ENDIF
#

      ENDPARDO a, a1, a2
#
#
      execute sip_barrier

#
                              ENDSIAL CCSD_RHF_SV1
```

Feasible computation using new language features Now, we give a new version of this code using the language new features introduced. The changes to the above are indicated with a trailing ##..

```
SIAL CCSD_RHF_SV1

      moaindex i= baocc, eaocc
      moaindex j= baocc, eaocc
      moaindex k= baocc, eaocc

      moaindex i1= baocc, eaocc
      subindex ii of i1    ##

      moaindex j1= baocc, eaocc
      subindex jj of j1    ##

      moaindex k1= baocc, eaocc

      moaindex a = bavirt, eavirt
      moaindex a1= bavirt, eavirt
      moaindex a2= bavirt, eavirt
      moaindex a3= bavirt, eavirt

      served VSaaai(a,a1,a2,i1)
      served TSaiai(a3,i1,a1,j1)

      temp TSaiaiSub(a3,ii,a1,jj)    ##
```

```
      temp taaaiii(a,a1,a2,k1,ii,jj)     ##

# ------------------------------------------------------------------------
# BEGIN MAIN PROGRAM
# ------------------
#
#
# Read transformed integrals from lists
# -------------------------------------
#
      CALL READ_2EL #standard code for this proc omitted

      PARDO a, a1, a2

            IF a  <= a1
            IF a1 <= a2

# First do the expensive part involving 3-virtual integrals and
#    of order VVVVooo
# ------------------------------------------------------------------------

            DO a3
            DO k1

               REQUEST VSaaai(a,a3,a2,k1) a

               DO j1

                  IF j1 <= k1

                     DO i1

                        IF i1 <= j1

                           REQUEST  TSaiai(a3,i1,a1,j1) a3

                           DO jj in j1    ##
                           DO ii in i1    ##

       #Now, the blocks of t1aaaiii are smaller because
       #  we have used subindices on the fourth and fifth dimension.
       #
       #Note that the contraction is just an ordinary contraction, no special
       #  machinery is needed.
       #
```

```
       #If we allow slices in arbitrary contexts, then the (explicit)
       #  introduction of TSaiaiSub is not needed and the contraction
       #  would simply be
       #  VSaaai(a,a3,a2,k1)*TSaiai(a3,ii,a1,jj)


                           TSaiaiSub(a3,ii,a1,jj) = TSaiai(a3,ii,a1,jj)     ##
                           t1aaaiii(a,a1,a2,k1,ii,jj)=                      ##
                                 VSaaai(a,a3,a2,k1)*TSaiaiSub(a3,ii,a1,jj)  ##

                           ENDDO ii
                           ENDDO jj
                        ENDIF
                     ENDDO i1
                  ENDIF
               ENDDO j1

            ENDDO k1

            ENDDO a3
#
# ------------------------------------------------------------------------
#
            ENDIF
            ENDIF
#

      ENDPARDO a, a1, a2
#
#
      execute sip_barrier

#
                           ENDSIAL CCSD_RHF_SV1
#
```

PROPOSED

3.4.2 PARDO with processor-groups *

Index of proposed changes with author 1.1

To write code that exhibits maximum parallelism, it is often useful to be able to indicate to the runtime environment that a certain PARDO construct should be given a limited

Work	SIAL code	G1	G2	G3	G4
	pre_pgroup 9 NG	*	*	*	*
	# NG is now 4				
	set_pgroup 1	Y	N	N	N
10 %	pardo i,j,k	*	-	-	-
10 %	pardo l,m,n	*	-	-	-
	set_pgroup 2	N	Y	N	N
30 %	pardo a,b,c	-	*	-	-
	set_pgroup 3	N	N	Y	N
30 %	pardo p,q,r	-	-	*	-
	set_pgroup 4	N	N	N	Y
20 %	pardo t,u,v	-	-	-	*
	set_pgroup 0	Y	Y	Y	Y
	sip_barrier	*	*	*	*

Table 8: A simple program using proces groups.

number of processors, rather than that it should try the standard work distribution or dynamic load balancing.

Then a PARDO will be executed by a group of processors, using any defined form of work distribution within the group for all index combination on the PARDO statement. Then, provided that there is no synchronization after the PARDO, the runtime environment can assign remaining processors, outside of that group, to execute the code after the PARDO. This way multiple PARDOs can be executing on multiple groups of processors in parallel. Each PARDO, for example, may be known to scale well up to 1,000 cores. Then a code with 10 PARDOs, will scale well up to 10,000 cores.

To define the appropriate groups an estimate of the work in the code with the set of PARDOs must be made. This is similar to the performance model and we can plan to use the performance model to generate this information during the dry-run phase of the execution. On small numbers of cores the groups would all become the group of all workers.

The information about the group and their relative size needs to be conveyed in some way by the programmer. We can think of this feature as a fully automated one, but that may lead to unexpected performance results. It seems desirable to allow the programmer to either express this in SIAL or to provide hints to the compiler and the runtime system through pragmas. This is a well-known technique familiar to programmers.

To focus the discussion, we explain a proptotype if these ideas that is being implemented in the existing SIAL and SIP environment. The application is the linear part of the perturbative triples code that is known to scale well because it has a lot of inherent parallelism. Here we sketch how we use processors groups to maximally expose this parallelism.

In Table 8 we give an overview of the work (vertical axis) and the processors (horizontal axis) involved in a SIAL program that uses process groups. A "*" indicates that the

processor is working, a "-" indicates that it skips the code block.
The argument "9" to the spcial super instruction pre_pgroup indicates the "triples formation" of groups, which is coded in the instruction. (The word formation is used here as in a formation of birds or aiarplanes.) The instruction has a section of code that knows how to set up the process groups for a given formation identifier, which is a number set by convention. The number 9 is chosen arbitrarily to indicate the triples formation. The second argument receives the number of process groups that the instruction has defined. The instruction takes into accoun the total number of available processors, as well as the size of the problem. This is written by hand in the example implementation, but could be the result of the performance model.
In the example pre_pgroup returns that 4 processor groups have been set up. Each processor group has a number of workers determined to enuser that the work given to each group will take about the same time, so that all groups run into the barrier at about the same time.
Note that before the barrier is called, the set_prgoup 0 sets the group back to the group zero of all workers.
It is important in this example implementation to realize that processor group require a section of code that consists on only PARDO blocks without any intervening serial code, except set_pgroup calls. This is required because only the PARDO instruction has been modified to pay attention to the processor group infrastructure and skip execution of the PARDO body when the processor is not a member of the active group.
The actual perturbative triples code fragment looks a bit more complicated than is shown in the table. The index IG runs over process groups. The special super instruction set_ijk defines ranges for three nested do-loops that perform the work. The index IT runs over the complete set of value triples for i,j,k and the code in the body of the do-loop over IT computes the necessary values for i, j, and k for each value of IT. The code inside the special superinstructions pre_pgroup for group formation number 9 and the code inside set_ijk must agree on how to divide up the work.

```
execute pre_pgroup 9 NG  # NG is now 4
do IG=1,NG
  execute set_pgroup IG
  pardo a,b,c
    execute set_ijk IG NT
    do IT=1,NT
      # do a batch of work in a merged threefold do
      # with ranges set by set_ijk
    enddo
  endpardo
enddo
execute set_pgroup 0
execute sip_barrier
```

PROPOSED

3.4.3 PARDO with grouping *

Index of proposed changes with author 1.1
Currently SIAL offers two types of loops: pardo, and do. For example

```
pardo i,j
   S0(i,j)
   do k
       S1(i,j,k)
        do m
             S2(i,j,k,m)
        enddo m
        S3(i,j,k)
    enddo k
     S4(i,j)
endpardo i,j
```

where S0(i,j) is some sequence of statements that may depend on indices i,j, etc. and where the iterations are independent. The iteration space of the pardo loop, including the nested do loops is $(i, j, k, m) : 1 \leq i \leq i^u, 1 \leq j \leq j^u, 1 \leq k \leq k^u, 1 \leq m \leq m^u$ where i^u is the declared upper bound of index i, etc. and for simplicity, assume that the lower bound of all indices is 1.
When this pardo loop is executed, the iteration space is partitioned so that the minimal worker task is (i,j,1:ku,1:mu). In other words, for each dimension, we have either a single value if the index is given in the pardo statement, or the entire declared range of the index if it is given in the (sequential) do statement. A particular index might be chosen by the programmer to execute in a sequential loop because it reduces communication in the particular computation or because it ensures that the granularity of the minimal task will be large enough.
In practice, the master is likely to assign a chunk of work containing multiple minimal tasks over some range of values of pardo indices (e.g. i and j) and the SIP implementation executes these as (implicit) sequential loops over the assigned values of i and j. The pardo loop would actually be executed in the SIP with something like

```
do  i=i0,i1     !i0,i1 chosen by master for load balance, i0=i1 possible
   do j=j0,j1   !j0,j1 chosen by master for load balance, j0=j1 possible
       S0(i,j)
       do k = k0,k1   !k0=1, k1=ku, entire declared range of k
            S1(i,j,k)
            do m = m0,m1   !m0=1,m1 = mu, entire declared range of k
                S2(i,j,k,m)
            enddo m
```

```
        S3(i,j,k)
    enddo k
    S4(i,j)
enddo j
enddo i
```

It is desirable to allow more flexible ways to partition the iteration space along a dimension. Instead of choosing a single value (if the index is determined by a pardo) or entire declared range (if the index is determined by a do), allow it to be partitioned into segments of some intermediate size.
The most straightforward approach (easiest for the compiler writer, and requiring only minor extensions in the sip to the mechanisms already in place) is to modifying the syntax of the pardo loop to allow an indication that certain indices should be more coarsely partitioned, and also introducing a new form of the do statement (stripdo) that nests inside the pardo. The former maintains the property that parallel execution is specified in a single place while the latter allows the programmer to maintain control over how the loops are nested. The parameter given to the strip modifier in the pardo indicates how many segments that dimension should be partitioned into. It is intended that this value be a hint rather than a strict value.

```
pardo i,j,strip(m,10)  !i,j partitioned as before, but m partitioned into
   S0(i,j)
   do k
       S1(i,j,k)
        stripdo m
             S2(i,j,k,m)
        endstripdo m
        S3(i,j,k)
    enddo k
     S4(i,j)
endpardo i,j,strip(m,10)
```

with execution in the SIP as

```
do  i=i0,i1     !i0,i1 chosen by master for load balance, i0=i1 possible
   do j=j0,j1   !j0,j1 chosen by master for load balance, j0=j1 possible
       S0(i,j)
       do k = k0,k1   !k0=1, k1=ku, entire declared range of k
            S1(i,j,k)
            do m = m0,m1   !m0, m1 chosen by master, approximate size is mu/10
                S2(i,j,k,m)
            enddo m
        S3(i,j,k)
    enddo k
    S4(i,j)
```

```
enddo j
enddo i
```

Note that the first example could also be specified as

```
pardo strip(i,iu)strip(j,ju), strip(k,1), strip(m,1)
stripdo i
stripdo j
   S0(i,j)
   stripdo k
       S1(i,j,k)
        stripdo m
             S2(i,j,k,m)
        endstripdo m
        S3(i,j,k)
     endstripdo k
     S4(i,j)
endstripdo j
endstripdo i
endpardo strip(i,iu)strip(j,ju), strip(k,1), strip(m,1)
```

where strip(i,iu) specifies the i dimension is broken into iu segments each of size one, and strip(k,1) indicates that the k dimension has only one segment that thus must contain the entire range.
However, it is not clear how this would interact with load balancing.

3.5 Operation statements

Any operation statement can include one or more valid array references, This means that

1. the array has been declared
2. each index has been declared
3. the type of each index used is the same as the type of the matching index in the declaration of the array
4. the range of each index used is a subrange of the range of the matching index in the declaration of the array

Any array reference that violates these conditions generates an assembly error. The assembler uses the predefined relationships between predefined constants to determine whether the range of an index is a subrange of the range of another index.

1. operations: +, -, *, ^ ==, <, >, <=, >=, && (and), || (or), ! (not)
2. operation-assignments: +=, -=, *=

3. allocate v3(mu,*,lambda,*)
   allocates all blocks for arrays declared as local; the blocks with the matching indices are allocated and are then available for processing; the allocate statement allows the user to specify partial allocation by listing the index explicitly, implying that only blocks with the (segment) value of the index at the time the allocate statement is executed will be allocated; specifying an index as the wildcard "*" allocates blocks for all values of the matching index as defined in the declaration of the local array; an allocate on any other array is an error.

4. deallocate v3
   deallocates all blocks for arrays declared as local; if no allocate has been executed for the local array when deallocate is executed, the deallocate is an error.

5. create v3
   allocates all blocks for arrays declared as distributed; in each task the blocks with the correct indices, e.g. as assigned by the master task to each task, are allocated and are then available for local processing; a create on any other array is an error.

6. delete v3
   deallocates all blocks for arrays declared as distributed; if no create has been executed for the distributed array when delete is executed, the delete is an error.

7. destroy v4
   removes all blocks of served array v4 from the disk storage and memory buffers of all IO servers. Served arrays do not have blocks associated with them until a prepare is issued for each block. Thus at any point in time a served array may have significantly less storage associated with it than the maximum implied by the declaration. Thus sparse arrays are automatically stored in a sparse way. This instruction allows the programmer to completely remove all storage associated with a served array. Then a new series of prepares can be started, possibly using less storage than the previous cycle.

8. v3(p,q,r,s) = v2(p,q,r,mu) * c(mu,s)
   is an assignment and a contraction. If the shape of the arrays does not match the contraction an assembly error will result. The order of the indices on the left and the right of the assignment do not have to be the same.

9. v3(p,q,r,s) = x(p,q) $\hat{y}$(r,s)
   is an assignment and a tensor product. If the shape of the arrays does not match the contraction an assembly error will result.

10. v3(p,q,r,s) += a * v1(p,q,r,s)
    multiply v1 by a scalar a and add the result to v3. The order of the indices on the left and the right of the assignment do not have to be the same.

11. v3(p,q,r,s) = a * v1(p,q,r,s)
    multiply v1 by a scalar a. v1 and v3 can be the same array. The order of the indices on the left and the right of the assignment do not have to be the same.

12. v3(p,q,r,s) *= a
    multiply v3 by a scalar a.

13. put v3(p,q,r,s) = v2(p,q,r,s)
    sends the local block of v2 to the owner task of the indicated block of the distributed array v3 to replace the existing block of v3. The shape and sizes of the blocks must match, as well as the order of the indices.

14. put v3(p,q,r,s) += v2(p,q,r,s)
    sends the local block of v2 to the owner task of the indicated block of the distributed array v3 to be accumulated there to the existing block of v3. The shape and sizes of the blocks must match, as well as the order of the indices.

15. get v3(p,q,r,s)
    gets the indicated block of a distributed array v3 from the owner task.

16. prepare v4(p,q,r,s) = v2(p,q,r,s)
    deliver a block of v2 to the server to replace the block of the served array v4 for future requests. The shape and sizes of the blocks must match, as well as the order of the indices.

17. prepare v4(p,q,r,s) += v2(pqp,r,s)
    deliver a block of v2 to the server to be added to the block of served array v4 for future requests. The shape and sizes of the blocks must match, as well as the order of the indices.

18. request v(mu,nu,lambda,sigma) sigma
    for served array v request the block with indicated indices and indicate that the next request will be for the listed index incremented by one, i.e. sigma+1. This will allow the SIP to anticipate and start communication that will most likely occur in the next iteration of the code, typically a loop.

19. prequest t(mu,nu,I,j) = v(mu,nu,a,b)
    Partial request. The array v must have been previously prepared. Then the prequest instruction will retrieve a partial block of data (mu,nu,i,j) from the full block of (mu,nu,a,b). Indices i and j must be declared as "index". Indices a and b can be any index type. Care must be taken to insure that i and j will take on values that are a sub range of indices a and b.

20. collective a += b
    collective operation to add the local variable b from every task into the local variable a.

21. execute specinstr arg1 arg2 arg3
    executes the predefined special instruction specinstr with arguments arg1 arg2 arg3. See Sect. 6 for a complete list and Sect. 3.5.3 for details on how to write a special super instruction and make it available for use in SIAL programs.

PROPOSED

3.5.1 Parallel library calls *

Index of proposed changes with author 1.1
The execution of a call to an external parallel library like ScaLAPACK is not a super instruction, since it will involve communication. Thus it is not suitable to make a call to an external parallel library like ScaLAPACK with the execute operation statement. A new operation statement needs to be defined to clearly indicate the nature of such library calls and distinguish them from the execution of special super instructions.

1. execute_parallel parallel_routine arg1 arg2 arg3
   executes the subroutine parallel_routine with arguments arg1 arg2 arg3 allowing and expecting that this routine will perform parallel operations with other tasks. See Sect. 4.3.6 for the need and an example use of this operation statement.

3.5.2 Synchronization operations

PROPOSED

Index of proposed changes with author 1.1
Currently sip_barrier and server_barrier are special super instructions listed in Sect 6. because they are collective communications they do not meet the strict definition of a super instruction, furthermore they are generic and should be an operation statement in the language.

1. sip_barrier
   syntax: sip_barrier
   function: causes the worker processors to synchronize. Must be used after distributed arrays are create, before distributed arrays are deleted, and in general whenever distributed arrays are used a barrier must be placed before the distributed array can be used.
   restrictions: none

2. server_barrier
   syntax: server_barrier
   function: causes the server processors to synchronize. Used in a manner similar to the way the sip_barrier is used when using distributed arrays except is relevant when served arrays are being used.
   restrictions: none

3.5.3 Super instructions

Super instructions are special compute kernels that perform operations on individual data blocks, sometimes called super numbers, with the constraint that all data references have to be to local memory. In other words, they cannot access remote data or perform any communication with remote tasks.

Special instructions were originally mainly used as temporary super instructions needed in development of SIAL language features. It provided a quick way to add an instruction without having to add it to the list of instructions supported by the SIAL compiler. However, it quickly grew into a way for SIAL programmers to add specialized code with compute kernels to a SIAL program that was specific to the particular algorithm, not intended for general use as a SIAL language feature.

Each special instruction has a call-back routine, which is called with a standard argument list. The argument list gives the instruction access to all tables used by the SIAL runtime, even though most instructions don't actually need all of these tables. Typically, the call-back routine is placed in the ACESIII/sip directory, although there is no requirement to do this.

A special instruction has the following syntax in the SIAL language:

```
execute special_instruction array1 array2
```

where special_instruction is the SIAL identifier for the instruction, and array1 and array2 are optional array arguments that the instruction may use.

To add the special instruction to the system, its call-back routine must be added to the list of instructions known to both the SIAL compiler and runtime code. To add the call-back to the compiler, an entry is added to the list in ACESIII/sip_shared/load_pre_defined_routines_compile_time.f. To add to the runtime, add an entry to the call-back routine in ACESIII/sip/load_pre_defined_routines.f. For example, if we were adding the special instruction "foobar", we would add the following line to ACESIII/sip_shared/load_pre_defined_routines_compile_time.f:

```
 dummy = load_user_sub('foobar'//char(0), 0)
```

We would also add the following code to ACESIII/sip_shared/load_pre_defined_routines.f:

```
external foobar
dummy = load_user_sub('foobar'//char(0), foobar)
```

Note that the calls to load_user_sub must occur in the same order at both compile time and runtime.

The structure of the subroutine that codes the actual compute kernel in the special super instruction must be wrapped in a routine that unpacks the arguments given by SIP. From these arguments, which are some index to the actual argument blocks in the block table, the address of the data can be found. This can then be used to pass to a routine that does the actual computation, such as a BLAS routine. An example super instruction that sums two blocks is shown in Sect. 8.4. The code shows how to unpack the arguments to get to the data inside the block. The example is written in Fortran, but the code looks similar in C/C++.

PROPOSED

3.5.4 Super instructions argument list *

Index of proposed changes with author 1.1
To support more special super instructions with more complex compute kernels in them, there is a need to pass more data blocks to the special instruction. The current argument limit is three. This needs to be extended to be a longer list, which may require a continuation line.

```
execute my_spec_si arg1 arg2 arg3 arg4 ....
```

To work with complex special super instructions in the current version, two extra special super instructions were written named set_ilist, for "internal list". Code then looks like this

```
# start building internal argument list
execute set_flags2 a1(i,j)
execute set_flags2 b2(mu,nu)
# execute the instruction
execute my_spec_si
```

The special set_ilist stored the location and properties of the argument, usually a block to be used by my_spec_si, in a global internal data structure that can be accessed by the special super instruction. The source code of the special super instruction setflags2 is shown in Sect. 8.5.
When the language is extended to accept a long list of arguments, the compiler can build the internal data structure. A convention must be set up to give the programmer of the special super instruction access to the arguments.
The fact that SIAL does not allow continuation lines and has a limit of 256 characters per line may pose a problem for building unlimited argument lists.

3.5.5 Super instruction for computing integrals

Currently the compute_integrals super instruction is defined in the language.

PROPOSED

Index of proposed changes with author 1.1
It is domain specific and should be a special instruction (see Sect. 6). This will be changed in the next version.

1. compute_integrals v(mu,nu,lambda,kappa)
   computes one block of integrals in the AO basis.

PROPOSED

3.6 Parallel sections *

Index of proposed changes with author 1.1

At the highest level of abstraction, SIAL currently provides a model of parallelism similar to what OpenMP calls "fork/join" parallelism. Parallelism is expressed explicitly with a pardo statement. Every pass through the code block of the pardo becomes a separate task and the tasks are dynamically mapped to processes by a master task. At the implementation level, the execution is SPMD. The sequential parts of the computation are actually replicated at all processes. It is usually the case (but this is not enforced by the SIAL semantics) that all processes performing these sequential parts will indeed perform the same computation at the SIAL level. In contrast to OpenMP, there is no implicit join at the end of a pardo loop. Instead, the SIAL programmer is expected to insert a sip-barrier instruction where it is needed. The next step is to allow a different form of parallelism to be expressed in SIAL programs. An obvious model is the parallel sections construct of OpenMP.

3.6.1 Informal syntax

A parsects-endparsects directive encloses a sequence of sections that could be executed in parallel. Sections are delimited by sect-endsect statements. Code that is not enclosed in a parsects directive belongs to an implicit parsects region that contains a single section.

```
...
parsects
sect
...
endsect
sect

...
endsect
endparsects
...
```

3.6.2 Grammar

```
<parallel section region> ::= parsects  <section>+  endparsects
               <section> ::=  sect   <codeblock>  endsect
<codeblock>::= SIAL code that may contain a sequence of pardo  blocks
```

3.6.3 Constraints

- pardo blocks may not contain a parsects region.
- parsects regions may not be nested.

3.6.4 Barriers

There are currently no implicit barriers in SIAL (in contrast with OpenMP). Keeping with that approach, there would be no implicit barriers related to parsects in SIAL, either. A possible semantics would be for a barrier appearing in a section to only apply to the processes executing the code in that section. (A barrier not enclosed in a section would apply to all processes).
Formally, a SIAL computation is a sequence of parsects regions (which may be implicit if the parsects region contains only one section). Each parsects region encloses a non-empty sequence of sections. A section may contain (non-nested) pardo loops. Barriers apply only within a section.

3.6.5 Allocating processors to sections

First, we give a brief description of the way this is done in OpenMP 3.0. OpenMP calls the thread available to a parallel region a team. The number of threads in a team can be set as an environment variable or by runtime routines. Setting the number of threads inside a section sets the number used for the next level. The number of threads available to the entire program can also be set. OpenMP allows arbitrary (but possibly implementation constrained) levels of nested parallelism.
In SIAL the most sensible place to specify the "number of processors in a team" would be at the top of the sect block. This would control the number of processes available for the pardo loops inside that section.

```
<section> ::=  sect (<team_size_desc> | e)  <block>  endsect
```

In SIAL, it is undesirable to indicate a specific number of processes in the source code. One can allow the programmer to give a qualitative (fuzzy) indication of appropriate number of processors as a fraction and let the SIP determine the actual values.
Example: Machine has 200 nodes.

```
parsects
sect 25\%          //results in 50 processes available for this section
     pardo ... endpardo
     pardo ... endpardo
endsect
sect 75\%      //results in 150 processes available for this section
     pardo ... endpardo
endsect
endparsects
```

4 Execution environment

4.1 SIP Components

Logically, SIP has the following components:

1. A component coordinating the work to be done by all tasks; this component executes in the master task during initialization;

2. A component for communication of basic data elements, blocks, between the cooperating tasks, the most visible aspect of this component is the distributed array;

3. A component for storing and retrieving large amounts of data, providing support for the served arrays;

4. A component for executing basic chunks of work in the form of super instructions; this component calls the communication and data storage components when necessary.

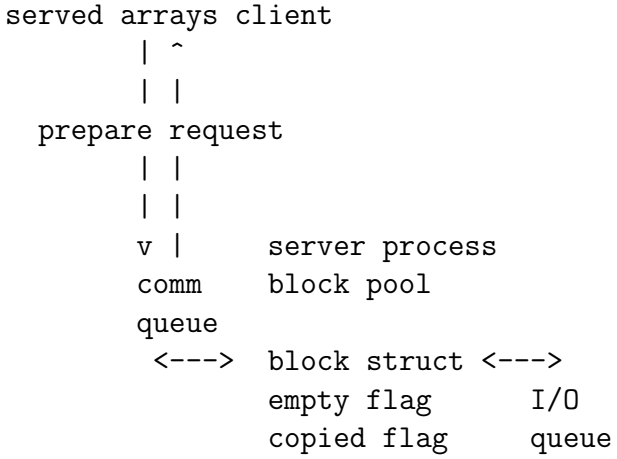
SIP is a parallel MPI program that consists of multiple tasks. Some tasks are dedicated to special functions, others are more general.

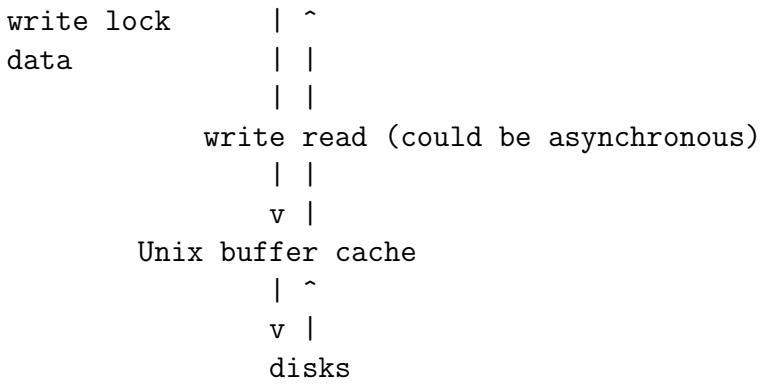
4.1.1 The IOCOMPANY

Tasks are grouped into companies, which perform a given function cooperatively. For example, one company, called the IOCOMPANY, is dedicated to providing support for served arrays. All data belonging to a served array is divided into blocks and blocks are always received into or sent from the memory of one of the tasks in the IOCOMPANY. The master task sets up tables designating which task will hold which block of every array using a simple algorithm, so that no searching for data blocks is necessary.

The algorithm is similar to that of managing paging space in a modern operating system. 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, which is caused by the need to restore the block from disk.

The operation of the IO servers are show in the diagram below:





1. The communication queue are the server front end and are processed at high priority, they are only blocked when all blocks in the pool are full and no blocks have the copy-to-disk-completed flag set to true.

2. The I/O queue are the back end and run at low priority, they copy blocks to disk, which means make system requests to copy blocks to the buffer cache and the kernel will move them to disk when it needed.

3. The block write lock is set whenever the front end updates a block with a prepare or the back end restores a previously deleted block from disk, and during that time all requests for the block are put in wait.

4.1.2 Worker companies

Other companies execute SIAL programs. All tasks in a company execute the same SIAL program. Very complex algorithms may require the cooperation of multiple companies, each executing a different SIAL program. The tasks in different companies can communicate with each other through the IOCOMPANY by reading and writing served arrays, for example. They can also communicate directly.

Communication of distributed array data is performed as follows: Each task has one or more threads running that are listening for requests. When a request is made the necessary locks are acquired to ensure integrity of the data, and then the block is asynchronously sent or received. While the communication is taking place, more requests for other blocks can be processed. Multiple requests to read the same block are also processed at the same time, thus reducing wait time for the client of the distributed array.

4.1.3 Super instruction processing

The activity of each task in all companies except the IOCOMPANY is controlled by a SIAL program. It is a list of super instructions to be executed. The super instructions can initiate communication, send or receive, or computation such as the tensor contraction of a two blocks of data into a third block. The computation can take a significant amount of time, depending on the block size. It is also possible that the instruction starts the computation of a block of integrals.

As much as possible, the super instructions are executed asynchronously: Communication operations are started and then control returns so that computation can be performed. When the data has is really needed, the task checks whether the communication instruction has completed successfully. The task can also look ahead in the instruction list and start certain operations early. The purpose of this flexibility is to try and maximize the hiding of communication delays behind computation work, thus minimizing over all waiting times in the execution of the parallel program.

PROPOSED

4.1.4 Executing super instructions on GPGPUs

Index of proposed changes with author 1.1
Initial work on executing super instructions inside a PARDO show promise of speed up. The main challenge is to keep control over the amount of data that must be transferred from CPU memory to GPU memory, since that is a slow operation.

4.2 Memory management

4.2.1 Data blocks and block stacks

SIAL is a language used to perform mathematical operations in parallel on arrays. In order to accomplish this, each index of an array is subdivided into segments, which in turn imposes a breakdown of the data into blocks. These blocks vary in size due to the types of indices comprising the array and the different segment sizes of those indices.

All memory used by the SIP is pre-allocated onto one of several different block stacks. Each block stack contains blocks of the same size. When a SIAL program is configured for execution, a pre-pass phase is executed, in which an estimate is made of the number of blocks needed on each stack as any instruction in the program is executed. Any remaining memory is allocated equally among the various stacks. As blocks are needed by the SIP instructions, they are allocated from the stack whose block size best fits the required block size. If all blocks on the desired stack are in use, the SIP tries to allocate a block from the stack with the next largest block size, and so forth.

Large data structures should be declared as DISTRIBUTED arrays, so that the blocks are spread over the memory of all tasks. All required communication to write and read blocks of distributed arrays is implicitly executed by the SIP. However, if a data structure does not have to be known to all tasks, then it is better to store just a part of the entire structure in an array declared as LOCAL. This way no communication is implied.

There is only one actual malloc done in the program ever. Not one per SIAL program, one malloc, period. This is done right up front, before a process ever knows that it is a worker or server. The amount of memory is given by the MAXMEM parameter, and the memory is never freed. It is simply retained until the end of execution of the entire program. It is reconfigured for reuse during the startup of each SIAL program, through calls to a memory

management subsystem that tracks the amount of the malloc memory. It was done this way when we put the joda stuff inside ACESIII. This way of working allows a processor to easily switch its usage of memory form joda's usage (before the SCF SIAL program runs), to the usage needed by a worker process (in SCF and possibly other SIAL codes), to a server's usage (if the process is allocated to the IOCOMPANY), then to the usage required at the end of a SIAL program (to set up buffers required to handle timing reports, gradient processing, etc.). It makes it all very dynamic, without the overhead of malloc/free calls.

To allow workers and IO servers to have a different amount of memory to work with, one would need to do the following:

1. Determine the MAXMEM parameter, as well as a new parameter, SERVER_MAXMEM, or whatever you think it should be called.

2. Change the malloc call to malloc the max(MAXMEM, SERVER_MAXMEM).

3. Add a new variable for use in the memory package, maxmem_available.

4. In the startup phase of each SIAL program, once it is determined whether a process is a worker or server, maxmem_available is set appropriately, to MAXMEM or SERVER_MAXMEM.

5. In the memory package, right now, there is an error if the total memory requested exceeds MAXMEM. This would be changed so that the error occurs if the memory requested exceeds MAXMEM_AVAILABLE.

The effect would be to allow the servers to use different amount of memory than the workers, but in reality, all you would be doing is forcing the workers to limit their use of memory to a lesser amount.

It is possible for the user to configure the amount of RAM for each tasks in such a way that a large amount of RAM is allocated to IO servers. This will effectively avoid all reads from disk for data because all PREPAREd blocks will be resident in the IO server cache for every REQUEST.

4.2.2 Memory estimate from a dry run

When a SIAL program is going through its startup phase, it initializes a number of tables and reserves memory for any static arrays needed during the calculations. After this, the program must determine the distribution of the remaining memory among the various block stacks, as well as whether it is actually possible to run the program with the remaining amount of memory on the number of processors provided. This is the role of the dry run.

The program calls a subroutine, stack_distribution, which in turn calls optable_loop_sim. Subroutine optable_loop_sim analyzes the SIAL instruction table and returns an array containing the number of blocks necessary for each type of block stack in order for the program to run. This array is used by stack_distribution to (1) determine if there is enough memory to actually hold this distribution of blocks, and (2) if so, distribute whatever excess memory may be present among the block stacks to come up with the final block distribution. If stack_distribution is able to come up with a valid distribution of blocks, the dryrun passes.

If the dryrun fails, the program goes into a loop to determine the minimum number of processors necessary to fit the job into this memory. During this loop, the number of trial processors is increased, the block_map_table is adjusted to redistribute the distributed and served blocks for each processor, and stack_distribution is called again with the trial setup, until a valid stack distribution is reached. This is the minimum number of processors. The user is notified, and the job aborts.

The key routine is optable_loop_sim, which goes through the instruction line by line, during the number of blocks needed for each stack at each SIAL instruction. During this analysis, no looping, iteration, or branching is done. The routine accounts for the number of blocks added by CREATEs, and ALLOCATEs, and subtracts blocks that are removed by DELETEs and DEALLOCATEs. Also, the routine correctly adjusts for temp blocks required during execution of loops. Optable_loop_sim gives a conservative, less-than-optimal estimate because it has no way of knowing the actual blocksize of the blocks used in a loop. Instead it uses the maximum segment size for each index of a block to determine its size, and therefore the appropriate block stack on which to place a block. If the block distribution it gives does fit into memory, it is guaranteed that the SIAL program will not run out of memory, if all else goes well.

Source code for stack_distribution is in the ACESIII/worker directory, optable_loop_sim is in ACESIII/sip.

4.2.3 Block stack management

All data is set up in arrays declared in the SIAL code. Arrays are declared either static, local, distributed, served, or temp. Each array is declared with the indices used in references to the array. The actual data of the array is broken down into blocks of one segment of each of the array's declared indices. The size of each segment of an index depends on the index type.

Static arrays are replicated on each processor in their entirety, and the memory to contain a static array is in the heap, but is reserved during the initialization of the SIAL program. Whatever heap memory remains after the static arrays are initialized is divided into memory stacks of different sizes, see Section 4.2.2 for details, and managed as blocks by the blkmgr. Blkmgr is simply a set of interface routines used to manage the processor's stacks of data blocks. Code for blkmgr is located in the file ACESIII/sip/blkmgr.F

The blocks of the arrays declared by SIAL are not all present at the same time, and they are only present when brought in and out of scope by SIAL instructions. For instance, a distributed array must be brought into scope by a CREATE instruction, and is removed by a DELETE instruction. Its blocks are scattered among the worker processes while it remains in scope, and it is accessible by GET and PUT instructions. When the array is DELETEd, its data blocks become available for use in other arrays.

Management of the block is performed by a set of flags, which are maintained for each block. When an instruction brings a block into scope, it must find a free block and reserve the block by setting the block_busy_flag. The routine performing this task is allocate_block. When a block is no longer needed, it must be freed, which is performed by free_block, which clears the flags associated with the block, releasing it for further use.

Temp blocks come into scope during the execution of SIAL instructions, such as the

result of a contraction, or as a block needed to hold remote data brought over in a GET or REQUEST instruction. When an instruction brings a temp block into scope, a flag indicating this is set in the instruction table. At the end of each DO or PARDO loop, some special code is executed to examine each instruction in the loop, and free the blocks if possible. This code is in ACESIII/sip/block_end_of_loop.f. If the block is still engaged in communication, this is not possible, so the block is put into a scrub queue, which is checked as blocks are needed later in the program. If the block is the result of a GET or REQUEST, it is not released, but flagged as a persistent block, which will not be reallocated by the algorithm in allocate_block unless absolutely necessary. Otherwise the block is freed. If a new GET or REQUEST is done, and the data is found to be in a persistent block already in memory, no MPI communication is necessary, the instruction simply sets the flags necessary to bring the block back into scope.

When allocate_block cannot find any free blocks of the proper size, it first searches the scrub queue, doing MPI_TEST calls on each block in the queue until a block is found that has completed its communication. If the block has not been flagged as persistent, it is now available for reuse. If no such block can be found, the persistent block queue (a FIFO) is searched, and the first block of sufficient size is allocated. In this way, the blocks keep coming available for reuse, although communication overhead is possible if memory is very tight and the program is forced to keep doing extra MPI requests due to continuously eliminating the persistent blocks.

4.2.4 Domain specific memory management

The SIP does all input/output. For compatibility with the rest of ACES II, the JOBARC files is read in the beginning and updated at the end of any SIAL execution; similarly the LISTS file(s) are read and updated to allow serial modules to intermix with parallel modules controlled by SIAL. In the SIAL language, there are no explicit read or write operations defined. The SERVED arrays are to be used for very large arrays and they can be assumed to be written in an efficient way in parallel by the SIP.

4.3 Execution management

4.3.1 Role assignment to tasks

When tasks are designated to become workers or IO servers, the master does pays attention to association of the tasks with the nodes they are running on. The demands placed on the hardware by a worker task is different than that by an IO server, as the input/output requests may generate different hardware activity depending on the way the storage system is connected to the nodes, or different patterns of communication in the case of storage systems that provide a parallel file system using the same fast interconnect that is used for MPI communication. Thus some nodes with 2 CPUs and 12 cores may end of with a number of workers and IO servers such that the number of IO servers exceeds some basic capability of the node.

To address this a more elaborate distribution than simple assignment of worker and IO server roles to tasks is orchestrated by the master to ensure that IO servers and workers are

mixed more evenly over nodes, e.g. so that no more than one or two tasks per node become IO servers.

The master tries to map one process per node to the IOCOMPANY, and it uses the MPI hostname to discriminate between the various nodes by using the value returned by MPI_GET_PROCESSOR_NAME, which is supposed to be different for processors on different nodes. On some systems it returns the same name for all processors, and then this algorithm does not work properly.

The code is in ACESIII/framelib/build_pst.F, call to mpi_get_processor_name in line 80 puts the name in the variable "hname", which is declared as

```
CHARACTER*(MPI_MAX_PROCESSOR_NAME).
```

4.3.2 PARDO processing

The PARDO construct should be used to execute a chunk of code on the local parts of a distributed array. All indices to be looped over in this way must be listed on the PARDO statement, because PARDO constructs cannot be nested. The master assigns which ranges of the block indices will be executed by each worker.

The ALSO PARDO allows the programmer to construct multiple blocks of code that each by themselves allow distributed processing and also are independent so that the code blocks can be executed on different sets of processors. If the number of processors is too small, the ALSO PARDO code blocks will be executed in serial, one after the other. But the construct allows the SIP to schedule work more efficiently if resources are available.

The regular DO construct should be used to execute a chunk of code for the complete range of the index specified in the DO statement. The DO construct can be arbitrarily nested.

A few special instructions have been defined. They are a simple mechanism in the SIAL language to process a small number of types of operations for which it is not worthwhile to develop a fully defined syntax.

All indices listed on the PARDO statement are evaluated as to their range, possibly restricted by a logical expression is the where-clause is present, and the total list of possible combinations is then distributed by the master task to all worker task in a static fashion.

When dynamic load balancing is in use, the work is distributed in a way that is identical to "guided" in the OpenMP standard. The workers get a partial assignment and when their assignment is complete they contact the master for more work.

The following lists the operation of load-balancing pardo.

1. Data used in the PARDO load-balancing algorithm is stored in the instruction table itself. This allows the master to track ongoing execution of different PARDOs, some of which may be in entirely different states of execution. Before modifying any of the data in the PARDO instruction entry in the instruction table, a lock must be acquired, and released after the data is modified.

2. When a PARDO is executed, it must be initialized. A flag in the instruction table indicates whether or not the loop has been initialized. If the loop must be initialized, the following occurs:

3. The current instruction context (start_op, end_op) are pushed onto the loop context stack, along with the current state of the WHERE conditions.

4. New values for start_op and end_op are calculated. These are the index of the current PARDO instruction and its corresponding ENDPARDO instruction. These values are needed to enable the SIAL program to execute nested loops and other instructions which interrupt sequential instruction execution.

5. The PARDO timer data is unpacked from an entry in the instruction table, and the PARDO timer is started. Also, any new WHERE conditions are added to the existing set of WHEREs already in effect.

6. Subroutine lb_set_loop_incr_mapping is called. This routine examines the set of PARDO indices, determines max_batch, the number of batches of work that the PARDO will perform. The variable last_batch_processed is set to the max_batch +1, which forces later iterations of the PARDO to reset on the first batch to be processed.

7. Initialization is concluded, and the instruction initialization flag is set to 1.

8. On a normal pass of the PARDO (I. e., one in which no initialization is performed), the following occurs:

9. The variables next_batch and last_batch are obtained from data in the instruction table. These define the range of batches undergoing processing on the current processor.

10. If next_batch ¿ last_batch, or next_batch = 0, then the current processor must acquire a new batch. This is done by communication with the master process. On a non-master processor, the following takes place:

11. The processor acquires the PARDO lock.

12. The routine PARDO_loadb_get_next_batch is executed. This routine sends a batch request to the master, then blocks until the master responds with a message containing the new data for next_batch, last_batch. This data is stored in the current processor's copy of the instruction table. If there is no more work, the master returns next_batch = -1.

13. The PARDO lock is released.

14. The current processor now has current data for next_batch, last_batch.
Routine lb_get_next_loop_set is called to convert the next_batch value into an array of segment values, each segment corresponding to one of the PARDO indices.

15. The values of each PARDO index in the index_table are set to the corresponding segment value from step 3 above. The current instruction pointer (iop) is incremented by 1, which forces instruction processing to begin at the next instruction following the PARDO. The PARDO routine returns to the instruction table processing loop.

16. The master determines the set of batches as follows:
    Recall that the total number of batches is max_batch. 90% of max_batch is distributed in equal size chunks of 0.9*max_batch/np, where np is the number of worker processes. When these batches have been distributed, the size of each batch is tapered linearly to a minimum of 1. Once the complete amount of work has been distributed, when any processor requests a new batch, the master returns a -1. This is a signal that the worker process may do its end-of-loop processing and move to the instruction following the PARDO loop.

4.3.3 End of loop processing

When a processor determines that it will receive no more work from the master for a PARDO loop, it skips to the ENDPARDO instruction at the end of the loop. Then it examines the instruction table entries between the beginning and end of the PARDO/ENDPARDO to determine whether an instruction has brought into existence any blocks as a result of the instruction. If it has, and the block is not needed for communication with other processors, the block is freed (I. e. becomes available for reuse). Otherwise, the block is added to a scrub queue. At some later point in the program, when a block is needed and the program is unable to find one that is free, the scrub stack will be searched and the block will be tested to see if the communication has finished. If so, it is scrubbed, and becomes available at that point. Blocks that have been brought into existence as the result of a GET or REQUEST instruction will be flagged as persistent blocks. This means that, even though their use is over and they are not engaged in communication, the program should leave them alone if possible. The block allocation scheme only frees a persistent block if all other blocks are in use.

4.3.4 IO Server activity

The IOCOMPANY parameter sets up the tasks allocated for use as I/O servers. When ACESIII determines the allocation of MPI processes to companies, it attempts to set up one I/O server on each compute node to avoid I/O bottlenecks. The exception to this is the Blue Gene. On the Blue Gene, an equal number of servers are allocated to each set of 64 compute nodes. This is due to the Blue Gene architecture, which allocates one I/O node to every 64 compute nodes.

Once the servers are allocated, the master sends each server a portion of the block map table. The portion received by a server lists only the blocks of each array that are that server's responsibility to manage. This data is used to build the server's own tables.

The server's main table is called the server_table. This table contains information about each of the server's managed blocks. A second table is the server_msg table, containing the set of messages currently being processed by the server. The rest of the server's memory is divided into blocks, each block equal to the largest block the server will be asked to process.

After the server's tables and memory is initialized, the server enters a processing loop. From this point on, the server's action is as follows:

1. Probe for a message from a worker.

2. When a message is detected, and a free message buffer is available, receive the message into the message buffer (also referred to as a message node).

3. For each active message node, the server enters the message handler routine for that node's particular message type.

The server works on different types of messages: request, prepare, barrier, blocks_to_list, etc. Each type of message supported by the server has a subroutine associated with it, a message handler. Each message handler is set up to be event-driven. When a message first enters a message node, it attains the begin_state. When a handler has determined that a message's processing is complete, it sets the message state in the node to null_state. In between, the message can take on different states that are valid for the processing of that particular message. For example, a request message may move from begin_state to wait_for_send_state to wait_for_block_state to request_cleanup_state to null_state. Other messages go through different states, handled by their own message handler routine.

When step 3 above has completed, the server checks for any nodes that are in null_state. If it finds one, it moves it from the active set to the free set, then loops back to step 1. The server terminates its infinite processing loop when it receives a quit message, indicating the worker's master process is initiating a server takedown at the end of a SIAL program. At this time, if server statistics have been collected for a timing report, it collects the data in the IOCOMPANY's master task and sends it to the worker's master task before exiting.

All source code for the server is in the ACESIII/manager directory.

PROPOSED

4.3.5 Fault tolerance *

Index of proposed changes with author 1.1

During the of multi-task-parallelism design, it is relevant and maybe essential to keep fault tolerance in mind. We must provide ways to decide that a task group will never deliver on its assigned work because processors in the task group have failed.

For the design of a general fault tolerant execution model, we can look to modern fault-tolerant enterprise computing strategies: replication for data and virtual machines for execution. Check-pointing can be compared to the distant past where regular backup to tape was used to protect computer operations. When a major fault occurred the system was rebuilt from tape.

1. **Data replication** The design of SIP can be modified without changing SIAL to support dynamic replication of data. We can use RAID 1 replication of blocks. With each block having two owners that are both updated at the same time and from which one is selected for reading. A more complicated scheme with RAID 5 style parity is possible. Then a number of copies of each block is kept and a block with checksum data. Implementation of data replication requires taking into account the information about processors being part of a single node, so that SIP

can make sure copies are kept on different nodes. As it is likely that faults with cause entire nodes to be taken out of the working collective.

2. **Fault tolerant execution** With data taken care of in an online fashion, execution can be dealt with very easily. If a processor does not respond it is marked as offline and the work it was doing is assigned to another processor. This will require a very simple communication primitive that provides access to heart-beat information.

Anticipating the fault tolerance in the design of SIP will make us ready when MPI gets there and allow us to explore other communication mechanisms that already have support for detecting failed members.
Notes:

1. Blue Waters will provide coarse grained LoadLeveler check-pointing.
2. A recent workshop addressed fault tolerance http://www.teragridforum.org/mediawiki/index.php?title=2009_Fault_Tolerance_Workshop.
3. The FT-MPI API is defined by the HARNESS (Heterogeneous Adaptive Reconfigurable Networked SyStem) runtime system http://icl.cs.utk.edu/harness/
4. The MPI 3.0 standard activity has a Fault Tolerance Working Group http://meetings.mpi-forum.org/mpi3.0_ft.php

PROPOSED

4.3.6 ScaLAPACK interoperability *

Index of proposed changes with author 1.1
To allow SIAL and SIP to make use of existing parallel libraries such as ScaLAPACK, some work needs to be done. The data used by ScaLAPACK is a regular cyclic-block distribution for matrices. Each worker who will collaborate on a ScaLAPACK call must allocate some blocks on its stack that are big enough to hold these cyclic matrix blocks. A routine must be provided to transfer blocks of SIAL type into these blocks of ScaLAPACK type.

5 Software development environment

To aid development and debugging a SIAL environment has been created to support development of SIAL programs and to aid in debugging and tuning them. This environment is a

plugin for Eclipse, the standard opensource and extensible environment for development and debugging; see the URL http://eclispse.org for full documentation and source of Eclipse.

5.1 Eclispe IDE

An Eclipse Plugin to enable IDE assisted development of SIAL programs is in progress. The update site for it is http://www.cise.ufl.edu/ njindal/sial/updates. Check out this site for basic eclipse tutorials. Once installed,

- Choose File-¿New-¿Other.
- In the dialogue choose General-¿Project.
- Give the project a name.
- Right-click on the project in the "Package Explorer" view, choose New→Other, General→File.
- Give it a name and an extension of .sial. (eg ccsd.sial).
- A new empty file will be created and Eclipse will correctly recognize it as a SIAL file.

Do not worry about the red mark at the beginning. As you type a correct program, this should go away.

5.2 SIAL IDE Features

1. Syntax highlighting
2. Outline view for pardos, calls, procedure declarations, etc.
3. Source Folding
4. Analyses
   (a) Find (and mark) suboptimal contractions
   (b) Find (and mark) unused variables
   (c) Find (and mark) unused procedures
5. Refactorings - To do refactoring, select some code, right-click-¿Refactorings...
   (a) Improve suboptimal contractions
   (b) Comment or remove unused variables
   (c) Comment or remove unused procedures

5.3 Building or compiling SIAL programs

This feature is not yet supported.

You would still need to edit the file in the IDE, transfer it to a machine which has ACESIII "installed" and compile it there.

5.4 Running SIAL programs

This feature is not yet supported.

5.5 Performance analysis tools

execute trace_on—off
turn on or off tracing features listed by their keyword.

The SIP keeps track of a number of timers that record the execution of super instructions and the traffic of MPI messages. At the end of the execution, the data is collected and averaged. The output file then lists the average, standard deviation, minimum and maximum for each timed instruction together with the source line number in the SIAL program.

If a user wants a report on the timing of SIAL instructions, he can get it by setting TIMERS=YES in the ACESIII ZMAT file, in the *SIP section. This provides timing data on each SIAL program that is run. If timing data for only one SIAL program is required, the user sets TIMERS=¡sio filename¿ instead.

The program collects the data automatically through a set of interface routines. The source code for the routines is in ACESIII/framelib/timer.f. The program sets up timers for each significant SIAL instruction (branch instructions and re-indexing type instructions are not timed). Also, for each PARDO loop, timers for the entire loop and a timer for the block wait time in the loop are set. Block wait time is the time the program is forced to wait for a block that is being sent from a remote processor, whether it be a worker or server.

In order to set up a timer, the programmer calls the subroutine register_timer. This routine accepts a user-specified character string as an identifier for the timer, and returns an integer key that is later used to reference the timer. The keys for each timer are stored in the instruction table entry of the instruction for which the timer applies.

As a SIAL program executes a SIAL instruction, it pulls the instruction's timer key (if one exists) out of the instruction table. The timing of the instruction begins by calling subroutine timer_start with this key. Then the instruction is executed. The timer is stopped by calling update_timer with the same key. Update_timer takes the difference in wall-clock times of the timer_start call and the update_timer call and sums it into the internal timer data structure.

After the SIAL program is finished, each processor's timer data is sent to the master process and summed together with the data corresponding to the same timer identifier string. The average, minimum, maximum, and standard deviation of the values across the processors are computed. Then the timer report is printed. Source code for this is in ACESIII/main/timer_data.F.

In addition to the timers, interface routines for program counters have been provided. Counters are registered similarly to timers, by calling register_counter to get a key for a counter. They are updated by the routine increment_counter. This code is also in ACESIII/framelib/timer.f. The counter report comes out after the timer report. Four counters have been created for each line of SIAL code for which there is an instruction timer. These counters are GET, PUT, PUTINC, PARDOMSG, REQUEST, PREPARE, and PREPSUM. The report of counter data comes out automatically whenever a timer report is created.

6 Listing of special super instructions

Several special instructions have already been developed and tested. They can be used in any SIAL program. Consider an example use.

energy_denominator v(a,i,b,j)

In coupled-cluster calculations it is necessary to divide a quantity, say $T_{old}(a,i,b,j)$ by [e(i)+e(j)-e(a)-e(b)] where e(k) are the orbital eigenvalues. The instruction
energy_denominator array(a,i,b,j)
divides each element of array(a,i,b,j) within the block by [e(i)+e(j)-e(a)-e(b)] and locally replaces that block array(a,i,b,j) by its 'scaled' value.
Example:
The distributed array $T_{old}(a,i,b,j)$ is divided by [e(i)+e(j)-e(a)-e(b)] and the result put into the distributed array $T2_{new}(a,i,b,j)$

```
PARDO a, b, i, j
   GET Told(a,i,b,j)
   execute energy_denominator Told(a,i,b,j)
   PUT T2new(a,i,b,j) = Told(a,i,b,j)
ENDPARDO a, b, i, j
```

6.1 Generic special super instructions

The special super instructions listed are of general use.

1. copy_ff
   syntax: execute copy_ff array1 array2
   function: Copies the array2 into the array1 without regard to index type
   restrictions: array1 and array2 must be two-dimensional static arrays.

2. set_index
   syntax: execute set_index array1
   function: Sets the indices of a 4-d array in common block values. These indices are stored in the SINDEX common block.
   restrictions: array1 must be a 4-dimension array with simple indices.

3. eigen_nonsymm_calc
   syntax: execute eigen_nonsymm_calc array1 array2
   function: Calculates the eigenvalues and eigenvectors of a 2-d square matrix. The matrix does NOT have to be symmetric. The matrix is also diagonalized on output. Array1 is the diagonalized matrix and array2 is the matrix whose columns are the eigenvectors of Array1.
   restrictions: array1 and array2 must be two-dimensional static arrays.

4. check_dconf
   syntax: check_dconf array1 scalar1
   function: The largest(absolute value) element of array1 is found and output as scalar1

restrictions: array1 must be two-dimensional and scalar1 must be declared as a scalar in the sial program.

5. return_diagonal4
   syntax: execute return_diagonal4 array1 array2
   function: The diagonal elements of the array array1 are removed and the resulting diagonal array is output as array2. array1 is not modified by the instruction.
   restrictions: Both array1 and array2 must be four-dimensional.

6. return_diagonal
   syntax: execute return_diagonal array1 array2
   function: The diagonal elements of the array array1 are removed and the resulting diagonal array is output as array2. array1 is not modified by the instruction.
   restrictions: Both array1 and array2 must be declared as static arrays in the sial program, and be two-dimensional.

7. return_sval
   syntax: execute return_sval array1(p,q) scalar1
   function: The scalar scalar1 is set equal to the value of the array array1. The overall purpose is to pull out the (p,q) element of the array1 and set scalar1 equal to its value.
   restrictions: array1 must be two dimensional and scalar1 must be declared scalar in the sial program.

8. place_sval
   syntax: execute place_sval array1(p,q) scalar1
   function: The (p,q) element of array1 is set equal to scalar1.
   restrictions: array1 must be two-dimensional. scalar1 must be defined as scalar in the sial program.

9. square_root
   syntax: execute square_root scalar1 scalar2
   function: scalar1 is raised to the power scalar2. scalar1 = scalar1**scalar2
   The name is misleading since it does the more general power instead of only the square root. restrictions: scalar1 and scalar2 must be declared as scalars in the sial program. To be a renamed power in the new version.

10. apply_den2
    syntax: execute apply_den2 source(p,q) target(i,j)
    function: each element of the array source(p,q) is divided by the corresponding element of the array target(i,j). The array source contains the output.
    restrictions: the arrays source and target must be two dimensional arrays.

11. apply_den4
    syntax: execute apply_den4 source target
    function: each element of the array source(p,q,r,s) is divided by the corresponding element of the array target(i,j,k,l). The array source contains the output.
    restrictions: the arrays source and target must be four dimensional arrays.

12. remove_diagonal
    syntax: execute remove_diagonal array1 array2
    function: The diagonal elements of the array1 are removed and the resulting array is array2.
    restrictions: array1 and array2 must be two-dimensional static arrays.

13. set_flags
    syntax: execute set_flags array1(i1,i2,i3)
    function: Sets the indices i1,i2,i3 of a 3-d array in common block values for further use by other special super instructions. The compiler passes not only the array information but also the indices of the block indicated by the indices to the special super instruction.
    restrictions: array1 must be three-dimensional with simple indices.

14. set_flags2
    syntax: execute set_flags2 array1(i1,i2)
    function: Sets the indices i1,i2 of a 2-d array in common block values. The compiler passes not only the array information but also the indices of the block indicated by the indices to the special super instruction.
    restrictions: array1 must be two-dimensional with simple indices.

15. print_scalar
    syntax: execute print_scalar scalar1
    function: prints the value of the scalar1 to standard output.
    restrictions: scalar1 must be declared as a scalar in the SIAL program.

16. dump_block
    syntax: execute dump_block array1(p,q,r,s)
    function: Writes out information about the block of array1(p,q,r,s). The first,last,maximum,and minimum values of the block are written out and the sum of squares of all elements in the block.
    restrictions: array1 must be of dimension 6 or less.

17. array_copy
    syntax: execute array_copy array1 array2
    function: To copy the array1 into array2 COMPLETELY.
    restrictions: array1 and array2 must have the same dimensionality and index types.

18. blocks_to_list/write_blocks_to_list
    syntax: execute blocks_to_list array(p,q,r,s)
    syntax: execute write_blocks_to_list
    Write the blocks of array to a list file or write the blocks of all arrays. This makes the data available in succeeding SIAL programs. It writes a simple FORTRAN sequential unformatted binary file containing all data blocks, called BLOCKDATA, as well as an index file called BLOCK_INDEX, used to determine the data format for reading the data blocks in the second SIAL program. There must be a sip_barrier after each blocks_to_list execution.

Note: The current implementation of this does not write an actual ACES II list file.
Note: The current implementation uses serial IO or MPI-IO, but needs improvment in performance.
Note: The current implementation does not allow proper flexibility to selecting what data to save for subsequent processing. Maybe a standard file-format like HDF5 could help in making this feature more useful, more portable, and more performant? function: To write all blocks in an array to a file. To use this instruction properly you must do the following.

- execute sip/server_barrier
- execute blocks_to_list array_k for all arrays being written out
- execute write_blocks_to_list
- execute sip/server_barrier

restrictions: none

19. list_to_blocks/read_list_to_blocks
syntax: execute list_to_blocks array(p,q,r,s)
syntax: execute read_list_to_blocks
Read the blocks of array from a list file or of all arrays. The data must have been written by a **blocks_to_list** execution in a preceding SIAL program. It reads the BLOCKDATA and BLOCK_INDEX files described in the section on blocks_to_list. The order of each list_to_blocks execution must be the same as that of the blocks_to_list statements from the first job. Also, there should be a sip_barrier executed after each list_to_blocks.
Notes: See under blocks_to_list.
function: To read all files(lists) and put them into arrays(blocked) . To use this instruction properly you must do the following.

- execute sip/server_barrier
- execute list_to_blocks array_k for all arrays being read in.
- execute execute read_list_to_blocks
- execute sip/server_barrier

restrictions: The data being read in must match up perfectly with the data in the blocks_to_list/write_blocks_to_list from the previous sial program.

20. sip_barrier
syntax: execute sip_barrier
function: causes the worker processors to synchronize. Must be used after distributed arrays are create, before distributed arrays are deleted, and in general whenever distributed arrays are used a barrier must be placed before the distributed array can be used.
restrictions: none
To become an operation statement in the new version, see Sect. 3.5.

21. server_barrier
    syntax: execute server_barrier
    function: causes the server processors to synchronize. Used in a manner similar to the way the sip_barrier is used when using distributed arrays except is relevant when served arrays are being used.
    restrictions: none
    To become an operation statement in the new version, see Sect. 3.5.

6.2 ACES III domain specific super instructions

The special super instructions listed are domain specific for ACES III and may or may not be useful for others.

1. return_h1
   syntax: execute return_h1 h1
   function: Computes the one-electron integrals of type kinetic and nuclear attraction, sums them and returns them as h1.
   restrictions: h1 must be a two-dimensional array.

2. fmult
   syntax: execute fmult array1
   function: Each element of the two-dimensional array1(i,j) is scaled by the fock matrix diagonal element (i,i).
   larray1(i,j) = array1(i,j)*fock(i,i).
   restrictions: array1 must be a two-dimensional array.

3. read_grad
   syntax: execute read_grad array1
   function: The array1(i,j) is read in and summed into the gradient which is in a common block.
   restrictions: array must be declared with simple indices. i and j should range from 1-natoms and 1-3, but simple indices have segment sizes of 1.

4. energy_denominator
   syntax: execute energy_denominator array1
   function: divides each element of array1(a,i,b,j,...) by the denominator fock(i,i)+fock(j,j)+..-fock(a,a)-fock(b,b)-....
   restrictions: array1 must be two, four, or six dimensional.
   The indices of array1 should have the correct spin type. i.e. (a,i) → (alpha,alpha), (b,j) → (beta,beta), etc.. Although the instruction would execute properly even if this were not the case but care should be taken using this instruction in the manner.

5. energy_adenominator
   syntax: execute energy_adenominator array1
   function: divides each element of array1(a,i) by the denominator fock_alpha(i,i) - fock_alpha(a,a).
   restrictions: array1 must be two dimensional.

6. energy_bdenominator
   syntax: execute energy_bdenominator array1
   function: divides each element of array1(a,i) by the denominator fock_beta(i,i) - fock_beta(a,a).
   restrictions: array1 must be two dimensional.

7. energy_abdenominator
   syntax: execute energy_abdenominator array1
   function: divides each element of array1(a,i) by the denominator fock_alpha(i,i) + fock_beta(i,i) - fock_alpha(a,a) - fock_beta(a,a).
   restrictions: array1 must be two dimensional.

8. read_hess
   syntax: execute read_hess array1
   function: The elements of the four dimensional array array1 are summed into the Hessian which is in a common block. Note that the summation is only performed on processor 0.
   restrictions: array1 must be a four dimensional array with simple index types. It must be dimensioned as (natoms,3,natoms,3).

9. fock_denominator
   syntax: execute fock_denominator array1
   function: The elements of the array1(a,i,b,j) are divided by fock(i,i) + fock(j,j) - fock(a,a) - fock(b,b). Note that if the denominator is zero that element of the array is set to zero.
   restrictions: array1 must be 2 or 4 dimensional.

10. der2_comp
    syntax: execute der2_comp array1(m,n,r,s)
    function: The derivative integrals for the block (m,n,r,s) are computed and returned in arrays1. Note that set_flags2 must have been used to define which degree of freedom to take the derivative with respect to. i.e. atom and component.
    restrictions: array1 must be a 4-dimensional array with AO indices and the coordinate with respect to which the derivative is taken must have been set, probably by set_flags2.

11. fock_der
    syntax: execute fock_der array1(mu,nu)
    function: Computes the derivative of the fock matrix from only one-particle contributions T+NAI and returns it as array1. The degree of freedom to take the derivative with respect to, i.e. atom and component, must have been previously set, probably by set_flags2.
    restrictions: Array1 must be two-dimensional array with AO indices. The perturbation must have been set before fock_der is called.

12. overlap_der
    syntax: execute fock_der array1(mu,nu)
    function: Computes the derivative of the overlap matrix. The degree of freedom to take

the derivative with respect to, i.e. atom and component, must have been previously set, probably by set_flags2.
restrictions: Array1 must be a two-dimensional array with AO indices. The perturbation must have been set before fock_der is called.

13. scontxy
    syntax: execute scontxy array1
    function: The second derivative 1-electron overlap integrals are computed and contracted with the array1. Note that array1 is perturbation independent and that all perturbations are considered inside the instruction. The Hessian is updated internally as well.
    restrictions: array1 must be a two-dimensional array with AO indices.

14. hcontxy
    syntax: execute hcontxy array1
    function: The second derivative 1-electron kinetic and nuclear attraction integrals(i.e. fock matrix contributions) are computed and contracted with the array1. Note that array1 is perturbation independent and that all perturbations are considered inside the instruction. The Hessian is updated internally as well.
    restrictions: array1 must be a two-dimensional array with AO indices.

15. compute_integrals
    syntax: execute compute_integrals v(mu,nu,lambda,kappa)
    computes one block of integrals in the AO basis.
    restrictions: array1 must be a 4-dimensional array with AO indices.
    This super instruction is not special in the current version, but it will be in the new version. See Sect. 3.5.3.

16. compute_sderivative_integrals
    syntax: execute compute_sderivative_integrals array1(m,n,r,s)
    function: The second derivative of the two-electron integrals is computed and contracted with array1. The perturbations (atom,component,jatom,jcomponent) defining the derivative are looped over internally and the hessian is updated internally.
    restrictions: array1 must be a 4-dimensional array with AO indices.

17. removevv_dd
    syntax: execute removevv_dd array1 array2
    function: removes all doubly occupied indices from the array1 with array2 being the result of the array with the all doubly occupied indices removed. Applicable if array1 = array1(b,b1), b = virtual beta index.
    restrictions: array1 and array2 must be two-dimensional arrays and they must have beta_virtual indices. nalpha_occ ¿ nbeta_occ is required. Only used for ROHF codes.

18. removeoo_dd
    syntax: execute removeoo_dd array1 array2
    function: removes all doubly occupied indices from the array1 with array2 being the result of the array with the all doubly occupied indices removed. Applicable if array1

= array1(i,i1), i = occupied alpha index.
restrictions: array1 and array2 must be two-dimensional arrays and they must have alpha_occupied indices. nalpha_occ ¿ nbeta_occ is required. Only used for ROHF codes.

19. remove_xs
syntax: execute remove_xs array1 array2
function: Removes the singly occupied components of the array1 which must be of type array1(a,i) which → array1(a,i_nosingles)
restrictions: array1 and array2 must be two-dimensional arrays and they must have (a,i) indices. a/i → alpha_virtual/alpha_occupied. Only used for ROHF codes

20. remove_xd
syntax: execute remove_xd array1 array2
function: Removes the doubly occupied components of the array1 which must be of type array1(a,i) which → array1(a,i_nodoubles)
restrictions: array1 and array2 must be two-dimensional arrays and they must have (a,i) indices. a/i → alpha_virtual/alpha_occupied. Only used for ROHF codes

21. remove_ds
syntax: execute remove_ds array1 array2
function: Truncates the array1(i,i) to array1(i_nodoubles,i_nosingles)
restrictions: array1 and array2 must be two-dimensional arrays and they must have (i,i) indices. i → alpha_occupied. Only used for ROHF codes

22. remove_ss
syntax: execute remove_ss array1 array2
function: Truncates the array1(b,b) to array1(b_nosingles,i_nosingles)
restrictions: array1 and array2 must be two-dimensional arrays and they must have (b,b) indices. b → beta_virtual OR (i,i), i → alpha_occupied. Only used for ROHF codes

23. comp_ovl3c
syntax: execute comp_ovl3c array1
function: Computes the three center overlap integrals and returns them in array1.
restrictions: array must be a three-dimensional array with AO indices.

24. udenominator
syntax: execute udenominator array1
function: The array1 is divided by an energy denominator just as in energy_denominator. udenominator does not require that the denominator not go to zero as small elements or zero denominators are eliminated.
restrictions: array1 can only be a 2 or a 4 dimensional array.

25. copy_fock
syntax: execute copy_fock array1 fock
function: Copies array1 into the fock array and copies the diagonal elements into the

corresponding eigenvalue array which is predetermined.
restrictions: The fock array is predetermined so the name must be correct, fock_a or fock$b. array1 must be a 2-dimension array with the same indices as the fock array.

7 List of domain specific SIAL programs and ACES III capabilities

The following capabilities are provided for the computational chemistry electronic structure software ACES III:

1. Hartree-Fock (HF) self-consistent field calculations (SCF) can be performed with restricted spin (RHF), unrestricted spin (UHF), or restricted open-shell (ROHF) energies, analytic gradients and analytic Hessians of the energy with respect to the molecular geometry.
   (a) RHF has all electrons confined to appear in the wave function in pairs with one electron in the spin up state and the other electron in the spin down state but both occupying the same orbital.
   (b) UHF allows electrons with spin up to have different spatial, i.e. orbital, distribution from the electrons with spin down.
   (c) ROHF wave functions are meaningful for molecules with an odd number of electrons and treat the one odd-man-out electron separately and pair all other electrons up like in the RHF wave function.
2. Second order many-body perturbation theory (MBPT2) also known as second order Mller-Plesset (MP2) energies and analytic gradients and analytic Hessians.
3. Coupled cluster singles and doubles (CCSD) energies and analytic gradients. Perturbative triples CCSD(T) energies can be computed as well.
4. Configuration interaction singles and doubles (CISD) excited states.
5. Coupled cluster equation-of-motion (CC-EOM) excited state energies are available, as well as EOM ionization potentials (IP) and electron affinities (EA).

With the gradients, the equilibrium molecular geometry can be computed as the minimum of the electronic energy as a function of atomic coordinates. For the methods without analytic gradients, a geometry optimization is possible with fully numerical gradients. Harmonic molecular vibrational frequencies can be computed by finite difference methods using gradients or second order finite difference methods using energies when analytic Hessians are not available. Further details of available capabilities can be found be consulting the manual.

To be written: organize the list of programs and briefly describe each program, full list with author 1

1. ccsd_rhf_ao_sv1_diis5.sial
   purpose:

2. ccsd_rhf_ao_sv1_diis5_kraken.sial
   purpose:
3. ccsd_uhf_ao_dist1_diis5.sial
   purpose:
4. ccsd_uhf_ao_sv1_diis5.sial
   purpose:
5. ccsd_uhf_ao_sv2_diis5.sial
   purpose:
6. ccsd_uhf_dropmo.sial
   purpose:
7. ccsd_uhf_mo_dist1_diis5.sial
   purpose:
8. ccsd_uhf_mo_sv1_diis5.sial
   purpose:
9. ccsdpt_rhf_aaa_new.sial
   purpose:
10. ccsdpt_rhf_aaa_new_dist1.sial
    purpose:
11. ccsdpt_rhf_aab_new.sial
    purpose:
12. ccsdpt_rhf_aab_new_dist1.sial
    purpose:
13. ccsdpt_rhf_pp.sial
    purpose:
14. ccsdpt_rhf_pp_new4.sial
    purpose:
15. ccsdpt_uhf_aax_new.sial
    purpose:
16. ccsdpt_uhf_bbx_new.sial
    purpose:
17. ccsdpt_uhf_d2_p1.sial
    purpose:
18. ccsdpt_uhf_pp.sial
    purpose:

19. ccsdpt_uhf_sv1.sial
    purpose:

20. cis_rhf.sial
    purpose:

21. cis_uhf.sial
    purpose:

22. cis_uhf_mo.sial
    purpose:

23. cis_uhf_mo_ao.sial
    purpose:

24. default_jobflows
    purpose:

25. ea_eomcc.sial
    purpose:

26. eccsd_rhf.sial
    purpose:

27. eccsd_uhf.sial
    purpose:

28. eom_dea.sial
    purpose:

29. eom_dip.sial
    purpose:

30. eomccsd_density_rhf.sial
    purpose:

31. eomccsd_density_uhf.sial
    purpose:

32. eomccsd_uhf_ao.sial
    purpose:

33. eomccsd_uhf_mo.sial
    purpose:

34. etran_rhf.sial
    purpose:

35. expand_cc.sial
    purpose:

36. expand_ccsd_rhf.sial
    purpose:

37. expand_lincc.sial
    purpose:

38. expand_linccsd_rhf.sial
    purpose:

39. gradscf.sial
    purpose:

40. guess_scf_uhf_finish.sial
    purpose:

41. hbar_uhf_ao.sial
    purpose:

42. hbar_uhf_ao_eaeomcc.sial
    purpose:

43. hbar_uhf_ao_ipeomcc.sial
    purpose:

44. hbar_uhf_mo.sial
    purpose:

45. hess_rhf_mp2_seg.sial
    purpose:

46. hess_rhf_sv1_ao1.sial
    purpose:

47. hess_rohf_ao1_sv.sial
    purpose:

48. hess_rohf_dist.sial
    purpose:

49. hess_uhf_alt.sial
    purpose:

50. hess_uhf_mp2_seg.sial
    purpose:

51. hess_uhf_mp2_seg_1.sial
    purpose:

52. hess_uhf_mp2_seg_2.sial
    purpose:

53. hess_uhf_scf.sial
    purpose:

54. hess_uhf_sv1_ao1.sial
    purpose:

55. instab_scf_uhf_mo_ao.sial
    purpose:

56. ip_eomcc.sial
    purpose:

57. lambda_rhf.sial
    purpose:

58. lambda_rhf_dropmo_new.sial
    purpose:

59. lambda_rhf_posteom_4dens.sial
    purpose:

60. lambda_uhf_ao_dist1_diis5.sial
    purpose:

61. lambda_uhf_ao_sv1_diis5.sial
    purpose:

62. lambda_uhf_ao_sv1_new.sial
    purpose:

63. lambda_uhf_ao_sv2_diis5.sial
    purpose:

64. lambda_uhf_dropmo.sial
    purpose:

65. lambda_uhf_dropmo_new.sial
    purpose:

66. lambda_uhf_mo_dist1_diis5.sial
    purpose:

67. lambda_uhf_mo_sv1_diis5.sial
    purpose:

68. lambda_uhf_posteom_4dens.sial
    purpose:

69. lccsdpt_rhf_aaa.sial
    purpose:

70. lccsdpt_rhf_aab.sial
    purpose:

71. linccsd_rhf_ao_sv1_cg.sial
    purpose:

72. linccsd_rhf_dropmo.sial
    purpose:

73. linccsd_uhf_ao_dist1_cg.sial
    purpose:

74. linccsd_uhf_ao_sv2_cg.sial
    purpose:

75. linccsd_uhf_dropmo.sial
    purpose:

76. mp2_n6_rhf.sial
    purpose:

77. mp2_n6_uhf.sial
    purpose:

78. mp2_rhf_sp.sial
    purpose:

79. mp2_uhf_ls1.sial
    purpose:

80. mp2_uhf_sp.sial
    purpose:

81. mp2grad_rhf.sial
    purpose:

82. mp2grad_rhf_sv1.sial
    purpose:

83. mp2grad_rhf_sv1_new.sial
    purpose:

84. mp2grad_rohf.sial
    purpose:

85. mp2grad_rohf_alt.sial
    purpose:

86. mp2grad_uhf.sial
    purpose:

87. mp2grad_uhf_sv1.sial
    purpose:

88. mp2grad_uhf_sv1_new.sial
    purpose:

89. non.sial
    purpose:

90. one_grad_herm_rhf_ao_sv1.sial
    purpose:

91. one_grad_herm_rhf_ao_sv1_dropmo.sial
    purpose:

92. one_grad_herm_uhf_ao_sv1_diis5.sial
    purpose:

93. one_grad_herm_uhf_ao_sv1_dropmo_diis5.sial
    purpose:

94. one_grad_rhf_ao_sv1.sial
    purpose:

95. one_grad_rhf_ao_sv1_dropmo.sial
    purpose:

96. one_grad_uhf_ao_dist1_diis5.sial
    purpose:

97. one_grad_uhf_ao_sv1_diis5.sial
    purpose:

98. one_grad_uhf_ao_sv1_dropmo_diis5.sial
    purpose:

99. one_grad_uhf_mo_dist1_diis5.sial
    purpose:

100. one_grad_uhf_mo_sv1_diis5.sial
     purpose:

101. rccsd_rhf.sial
     purpose:

102. rccsdpt_aaa.sial
     purpose:

103. rccsdpt_aab.sial
     purpose:

104. rccsdpt_sub_aaa.sial
purpose:

105. rccsdpt_sub_aab.sial
purpose:

106. reom_rhf.sial
purpose:

107. reom_rhf_printvec_4dens.sial
purpose:

108. reom_uhf.sial
purpose:

109. reom_uhf_printvec_4dens.sial
purpose:

110. rohf_tran.sial
purpose:

111. scf_aguess.sial
purpose:

112. scf_ccsd_rhf.sial
purpose:

113. scf_comp_rud.sial
purpose:

114. scf_rhf_aguess.sial
purpose:

115. scf_rhf_fast.sial
purpose:

116. scf_rhf_isymm_diis10.sial
purpose:

117. scf_rhf_new.sial
purpose:

118. scf_rud_model.sial
purpose:

119. scf_uhf_aguess.sial
purpose:

120. scf_uhf_fast.sial
purpose:

121. scf_uhf_finish.sial
purpose:

122. scf_uhf_init.sial
purpose:

123. scf_uhf_isymm_diis10.sial
purpose:

124. scf_uhf_new.sial
purpose:

125. sial_config
purpose:

126. tran_rhf_ao_sv1.sial
purpose:

127. tran_rhf_expanded.sial
purpose:

128. tran_rhf_expanded_lincc.sial
purpose:

129. tran_uhf_ao_dist1.sial
purpose:

130. tran_uhf_ao_sv1.sial
purpose:

131. tran_uhf_expanded.sial
purpose:

132. tran_uhf_expanded_lincc.sial
purpose:

133. tran_uhf_mo_dist1.sial
purpose:

134. tran_uhf_mo_sv1.sial
purpose:

135. tran_uhf_vvvv.sial
purpose:

136. two_grad_herm_rhf_ao_sv1.sial
purpose:

137. two_grad_herm_uhf_ao_sv1.sial
purpose:

138. two_grad_herm_uhf_ao_sv1_dropmo.sial
purpose:

139. two_grad_rhf_ao_sv1.sial
purpose:

140. two_grad_uhf_ao_dist1.sial
purpose:

141. two_grad_uhf_ao_sv1.sial
purpose:

142. two_grad_uhf_ao_sv1_dropmo.sial
purpose:

143. two_grad_uhf_mo_dist1.sial
purpose:

144. two_grad_uhf_mo_sv1.sial
purpose:

8 Example Programs

8.1 SIAL program using a procedure, a served array and a distributed array

```
SIAL example1
aoindex lambda=1,norb
aoindex sigma=1,norb
aoindex mu=1,norb
aoindex nu=1,norb
moindex p=bocc,eocc
moindex q=bocc,eocc
moindex r=bvirt,evirt
moindex s=bvirt,evirt
served v(mu,nu,lambda,sigma) # the SIP knows how to distribute
                             # the integral requests, it is not
                             # specified in the language since it
                             # can change every run
temp v1(p,nu,lambda,sigma)
temp v2(p,q,lambda,sigma)
temp v3(p,q,r,sigma)
distributed v4(p,q,r,s)
local c(mu,p)

PROC update
```

```
  # start new accumulate and checks on all outstanding ones
  # to make the SIP work efficiently several v4 = v3 * c must be allowed
  # to start so that of all accumulates in progress at least one
  # is ready every time accumulate is executed by the SIP
  put v4(p,q,r,s) += v4(p,q,r,s)
  return
ENDPROC update

create v4
pardo mu, nu
  do lambda
  do sigma
    request v(mu,nu,lambda,sigma) sigma
    # ask for an integral block
    # the first call initiates a request
    # subsequent calls check that at
    # least one of the outstanding
    # requests completed and
    # makes a new request
    # because this fetch happens outside a 4-fold loop most likely
    # one outstanding request is sufficient
    do p
      v1(p,nu,lambda,sigma) = v(mu,nu,lambda,sigma) * c(mu,p)
      do q
        v2(p,q,lambda,sigma) = v1(p,nu,lambda,sigma) * c(nu,q)
        do r
          v3(p,q,r,sigma) = v2(p,q,lambda,sigma) * c(lambda,r)
          do s
            v4(p,q,r,s) = v3(p,q,r,sigma) * c(sigma,s)
            call update
          enddo s
        enddo r
      enddo q
    enddo p
  enddo sigma
  enddo lambda
endpardo mu, nu
delete v4
ENDSIAL example1
```

8.2 SIAL program preparing a served array

```
SIAL example2
aoindex lambda=1,norb
aoindex sigma=1,norb
```

```
aoindex mu=1,norb
aoindex nu=1,norb
moindex p=bocc,eocc
moindex q=bocc,eocc
moindex r=bvirt,evirt
moindex s=bvirt,evirt
served v(mu,nu,lambda,sigma)
temp v1(p,nu,lambda,sigma)
temp v2(p,q,lambda,sigma)
temp v3(p,q,r,sigma)
temp v4tmp(p,q,r,s)
served v4(p,q,r,s)
local c(mu,p)
pardo mu, nu
  do lambda
  do sigma
    request v(mu,nu,lambda,sigma) sigma
    do p
      v1(p,nu,lambda,sigma) = v(mu,nu,lambda,sigma) * c(mu,p)
      do q
        v2(p,q,lambda,sigma) = v1(p,nu,lambda,sigma) * c(nu,q)
        do r
          v3(p,q,r,sigma) = v2(p,q,lambda,sigma) * c(lambda,r)
          do s
            v4tmp(p,q,r,s) = v3(p,q,r,sigma) * c(sigma,s)
            prepare v4(p,q,r,s) += v4tmp(p,q,r,s)
          enddo s
        enddo r
      enddo q
    enddo p
  enddo sigma
  enddo lambda
endpardo mu, nu
# Now the program can use v4 with request v4.
ENDSIAL example2
```

8.3 SIAL program using served arrays

Consider a parallelization scheme for integral transformation that Victor Lotrich has implemented in the UHF transformation code. This scheme basically narrows the range of the PARDO while at the same time contracting out an entire index on one processor, thereby making it possible to replace prepare +=’s with simple prepares.

Old style:

```
PARDO mu, nu, a, i
```

```
  #
  REQUEST Vxxai(mu,nu,a,i) i
  #
  DO a1
    #
    Txaai(mu,a1,a,i) = Vxxai(mu,nu,a,i)*ca(nu,a1)
    PREPARE Vxaai(mu,a1,a,i) += Txaai(mu,a1,a,i)
    #
  ENDDO a1
  #
ENDPARDO mu, nu, a, i
```

This loop distributes the parallelization over mu,nu,a,i in an effort to avoid re-reading the data in the REQUEST. However, this code is forced to use PREPARE +=, which is deadly on performance.

New style:

```
PARDO mu, a, i
  #
  ALLOCATE Lxaai(mu,*,a,i)
  DO nu
    REQUEST Vxxai(mu,nu,a,i) i
    #
    DO a1
      #
      T1xaai(mu,a1,a,i) = Vxxai(mu,nu,a,i)*ca(nu,a1)
      Lxaai(mu,a1,a,i) += T1xaai(mu,a1,a,i)
      #
    ENDDO a1
  ENDDO nu
  #
  DO a1
    PREPARE Vxaai(mu,a1,a,i) = Lxaai(mu,a1,a,i)
  ENDDO a1
  DEALLOCATE Lxaai(mu,*,a,i)
ENDPARDO mu, a, i
```

This loop reduced the PARDO range to mu,a,i, but a complete contraction of the nu index is performed for each (mu,a,i) combination. Thus we can do a PREPARE instead of PREPARE +=. Note that we still are reading the entire set of input only once. There is some wait time associated with the DEALLOCATE instruction until the PREPAREs are complete, but this is much smaller than going the PREPARE += route. There are 3 loops in this code that can be restructured with this same approach.

8.4 Special super instruction sum_64ss

The subroutine sums two blocks. The logic of unpacking the argument list into addresses that can be used for a call to the routine dosum_64ss that does the actual work is clearly shown.

```
C  Copyright (c) 2003-2010 University of Florida
C
C  This program is free software; you can redistribute it and/or modify
C  it under the terms of the GNU General Public License as published by
C  the Free Software Foundation; either version 2 of the License, or
C  (at your option) any later version.

C  This program is distributed in the hope that it will be useful,
C  but WITHOUT ANY WARRANTY; without even the implied warranty of
C  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
C  GNU General Public License for more details.

C  The GNU General Public License is included in this distribution
C  in the file COPYRIGHT.
      subroutine sum_64ss(array_table, narray_table,
     *                    index_table,
     *                    nindex_table, segment_table, nsegment_table,
     *                    block_map_table, nblock_map_table,
     *                    scalar_table, nscalar_table,
     *                    address_table, op)
c-----------------------------------------------------------------------
c   array1 --> 6D
c   array2 --> 4D
c   The last two indeces of array1 MUST be simple.
c-----------------------------------------------------------------------

      implicit none
      include 'interpreter.h'
      include 'trace.h'
      include 'mpif.h'
      include 'epsilon.h'
#ifdef ALTIX
      include 'sheap.h'
#endif

      integer narray_table, nindex_table, nsegment_table,
     *        nblock_map_table
      integer op(loptable_entry)
      integer array_table(larray_table_entry, narray_table)
      integer index_table(lindex_table_entry, nindex_table)
```

```
      integer segment_table(lsegment_table_entry, nsegment_table)
      integer block_map_table(lblock_map_entry, nblock_map_table)
      integer nscalar_table
      double precision scalar_table(nscalar_table)
      integer*8 address_table(narray_table)

      integer i, j, k
      integer array, index, nindex, ierr
      integer block, blkndx, seg
      integer find_current_block
      integer*8 indblk1, indblk2, get_block_index
      integer stack

      integer comm

      integer fop1(mx_array_index), fop2(mx_array_index)
      integer sop1(mx_array_index), sop2(mx_array_index)
      integer sindex(6), findex(4)
      integer type(mx_array_index)
      integer na1, na2, ni1, ni2
      integer*8 addr, get_index_from_base
      double precision x(1)
      double precision y(1)
#ifdef ALTIX
      pointer (dptr, x)
#else
      common x
#endif

#ifdef ALTIX
      dptr = dshptr
#endif

c-------------------------------------------------------------------------
c   Detrmine the parameters of the first array: c_result_array
c-------------------------------------------------------------------------

      array = op(c_result_array)
      nindex = array_table(c_nindex, array)

      do i = 1, nindex
         index   = array_table(c_index_array1+i-1,array)
         type(i) = index_table(c_index_type, index)
         seg     = index_table(c_current_seg,index)
```

```
         sindex(i) = index
         call get_index_segment(index, seg, segment_table,
     *                           nsegment_table, index_table,
     *                           nindex_table, sop1(i), sop2(i))
       enddo

       if (array_table(c_array_type,array) .eq. static_array) then
         addr    = address_table(array)
         indblk1 = get_index_from_base(addr, x, 2)
       else
         block = find_current_block(array, array_table(1,array),
     *                           index_table, nindex_table,
     *                           segment_table, nsegment_table,
     *                           block_map_table, blkndx)

         stack   = array_table(c_array_stack,array)
         indblk1 = get_block_index(array, block, stack,
     *                          blkndx, x, .true.)
       endif

c-------------------------------------------------------------------------
c   Detrmine the parameters of the second array: c_op1_array
c-------------------------------------------------------------------------

       array = op(c_op1_array)
       nindex = array_table(c_nindex, array)

       do i = 1, nindex
         index   = array_table(c_index_array1+i-1,array)
         type(i) = index_table(c_index_type, index)
         seg     = index_table(c_current_seg,index)

         findex(i) = index
         call get_index_segment(index, seg, segment_table,
     *                           nsegment_table, index_table,
     *                           nindex_table, fop1(i), fop2(i))
       enddo


       if (array_table(c_array_type,array) .eq. static_array) then
         addr    = address_table(array)
         indblk2 = get_index_from_base(addr, x, 2)
       else
         block = find_current_block(array, array_table(1,array),
     *                           index_table, nindex_table,
```

```
     *                              segment_table, nsegment_table,
     *                              block_map_table, blkndx)

         stack   = array_table(c_array_stack,array)
         indblk2 = get_block_index(array, block, stack,
     *                             blkndx, x, .true.)
      endif

c     write(6,*) ' ******************** '
c     write(6,*) ' OP1 :', (sindex(i), i=1,4)
c     write(6,*) ' OP2 :', (findex(i), i=1,4)

      call dosum_64ss(x(indblk1), sindex,
     *                  sop1(1), sop2(1), sop1(2), sop2(2),
     *                  sop1(3), sop2(3), sop1(4), sop2(4),
     *                  sop1(5), sop2(5), sop1(6), sop2(6),
     *                  x(indblk2), findex,
     *                  fop1(1), fop2(1), fop1(2), fop2(2),
     *                  fop1(3), fop2(3), fop1(4), fop2(4))

      return
      end

      subroutine dosum_64ss(x, sindex, a1, a2, b1, b2, c1, c2, d1, d2,
     *                                 s1, s2, ss1, ss2,
     *                         y, findex, e1, e2, f1, f2, g1, g2, h1, h2)
      implicit none
      include 'interpreter.h'
      include 'trace.h'
      include 'mpif.h'
      include 'epsilon.h'
#ifdef ALTIX
      include 'sheap.h'
#endif

      integer a1,a2,b1,b2,c1,c2,d1,d2
      integer e1,e2,f1,f2,g1,g2,h1,h2
      integer s1, s2, ss1, ss2
      integer fop1(mx_array_index), fop2(mx_array_index)
      integer sop1(mx_array_index), sop2(mx_array_index)
      integer sindex(6), findex(4)

      integer i, j, k, a, b, c, d
      integer m, n, lda, ldb, ldc
```

```
      double precision x(a1:a2,b1:b2,c1:c2,d1:d2,s1:s2,ss1:ss2)
      double precision y(e1:e2,f1:f2,g1:g2,h1:h2), xtemp

      do d = d1, d2
      do c = c1, c2
      do b = b1, b2
      do a = a1, a2

c        xtemp = y(a,b,c,d)
c        do i = s1, s2
c        do j = ss1, ss2

            x(a,b,c,d,s1,ss1) = x(a,b,c,d,s1,ss1) +  y(a,b,c,d) ! xtemp
c           x(a,b,c,d,i,j) = x(a,b,c,d,i,j) +  xtemp

c        enddo
c        enddo

      enddo
      enddo
      enddo
      enddo

      return
      end
```

8.5 Special super instruction set_flags2

The subroutine takes the indices from an argument array-block and stores them for use by a subsequent special super instructions.

```
C  Copyright (c) 2003-2010 University of Florida
C
C  This program is free software; you can redistribute it and/or modify
C  it under the terms of the GNU General Public License as published by
C  the Free Software Foundation; either version 2 of the License, or
C  (at your option) any later version.

C  This program is distributed in the hope that it will be useful,
C  but WITHOUT ANY WARRANTY; without even the implied warranty of
C  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
C  GNU General Public License for more details.

C  The GNU General Public License is included in this distribution
C  in the file COPYRIGHT.
```

```
      subroutine set_flags2(array_table, narray_table,
     *                      index_table,
     *                      nindex_table, segment_table, nsegment_table,
     *                      block_map_table, nblock_map_table,
     *                      scalar_table, nscalar_table,
     *                      address_table, op)
c-----------------------------------------------------------------------
c   Sets the indices of a 3-d static array in common block values.
c   The first index is assumed to be the atom, the second is the component
c   index (i. e. x,y, or z), and the 3rd is the center.
c
c   These indices are stored in the d2int_com common block, and are meant
c   to indicate the atom, component, and center on which to calculate a
c   single block of derivative integrals.
c
c   Example:
c      index jatom = 1, natoms
c      index jx = 1,3
c      static flags2(jatom, jx)
c
c      taoint(mu,nu,lambda, sigma) = 0.
c      do jatom
c      do jx
c         execute set_flags2 flags2(jatom, jx)
c         execute d2int aoint(mu, nu, lambda, sigma)
c         taoint(mu,nu,lambda, sigma) += aoint(mu, nu, lambda, sigma)
c      enddo jx
c      enddo jatom
c
c-----------------------------------------------------------------------
      implicit none
      include 'interpreter.h'
      include 'mpif.h'
      include 'trace.h'
      include 'parallel_info.h'

      common /d2int_com/jatom, jx, jcenter
      integer jatom, jx, jcenter
      double precision flags_value

      integer narray_table, nindex_table, nsegment_table,
     *        nblock_map_table
      integer op(loptable_entry)
      integer array_table(larray_table_entry, narray_table)
      integer index_table(lindex_table_entry, nindex_table)
```

```
      integer segment_table(lsegment_table_entry, nsegment_table)
      integer block_map_table(lblock_map_entry, nblock_map_table)
      integer nscalar_table
      double precision scalar_table(nscalar_table)
      integer*8 address_table(narray_table)

      integer ierr, array, array_type, ind, nind
      integer i

      array = op(c_result_array)
      if (array .lt. 1 .or. array .gt. narray_table) then
         print *,'Error: Invalid array in set_flags, line ',
     *     current_line
         print *,'Array index is ',array,' Allowable values are ',
     *      ' 1 through ',narray_table
         call abort_job()
      endif

      nind = array_table(c_nindex, array)
      if (nind .ne. 2) then
         print *,'Error: set_flags2 requires a 2-index array.'
         call abort_job()
      endif

c-----------------------------------------------------------------------
c   Atom, component, and center indices are determined from the c_current_seg
c   field of the 1st and 2nd index of the array.
c-----------------------------------------------------------------------

      ind   = array_table(c_index_array1,array)
      jatom = index_table(c_current_seg, ind)
      ind   = array_table(c_index_array1+1,array)
      jx    = index_table(c_current_seg, ind)

c-----------------------------------------------------------------------
c   Set jcenter to 0 as it is not currently being used.
c-----------------------------------------------------------------------

      jcenter = 0

      return
      end
```

9 Format of the .sio file

File type is binary unformatted. Each .sio file contains a 7-word header record followed by 4 tables, each of which are created by the compiler.

9.1 Header record

1. Id
   An identifier used to check that this is truly a .sio file (currently 70707).

2. Version
   Version identifier

3. Release
   Release identifier

4. Nindex_table
   Number of entries in the index table.

5. Narray_table
   Number of entries in the array table.

6. Noptable
   Number of entries in the operation table.

7. Nscalar_table
   Number of entries in the scalar table.

The header record is followed by the index table, array table, operation table, and scalar table, in that order. All data is integer, except for the scalar table, which is double precision. Items that are filled in at run-time will be denoted by RT.

9.2 Index Table

Consists of Nindex_table entries. Each entry has a structure defined in the file ACE-SIII/include/interpreter.h. The index table structure is as follows.

1. Index_size
   Length of the index in words. The compiler fills in an appropriate symbolic constant for each type of index, and the actual size is filled in at run-time when the true size is determined from job parameters.

2. Nsegments
   Number of segments (RT).

3. Current_seg
   Current segment to process. (RT).

4. Bseg
   Beginning segment of the index. (RT).

5. Eseg
   Ending segment of the index. (RT).

6. Index_type
   Type of index (aoindex=1001, moindex=1002, etc.)

7. Next_seg
   Used in loop operations to predict the next segment to process. (RT)

9.3 Array Table

Each array table consists of entries of integer data as follows:

1. Nindex
   Number of indices defined for the array.

2. Array_type
   Type of the array (i. e. static, distributed, served).

3. Numblks
   Total number of blocks in the array (RT).

4. Index_array1
   The following mx_array_index words contain the slots for the array's indices. These indices point to index_table entries, corresponding to the description of each index. The compiler inserts the indices used in the array definition, but the run-time fills in the actual indices used in references to the array at run-time as an instruction is processed.

5. Index_range1
   The following mx_array_index words are used to save the original indices used in the array definition. In some cases, this information is required to properly process an instruction, and the indices in the index_array1 field are possibly not the correct types due to being changed in the course of processing instructions (RT).

6. Block_map
   A pointer to the beginning of the array's block_map_table entries. (RT).

7. Scalar_index
   A pointer into the scalar table (used only for scalar values). Each scalar actually has an array table entry, with the array_type set to scalar_value (205). The scalar_index field of the array table points into the scalar table to the actual scalar value itself.

8. Create_flag
   Used to indicate that the array has entered scope via a create instruction (RT).

9. Put_flag
   Used to indicate that the array has had put instructions executed in the current loop with this array as target (RT)

10. Prepare_flag
    Used to indicate that the array has had prepare instructions executed in the current loop with this array as target (RT).

11. Current_blkndx
    Points to the current block of the array in the block manager's data structures. The current block is the block of the array corresponding to each of its indices having the current_seg value of the index_table. The current_blkndx is used to quickly reference this block without having to perform a search.

12. Array_stack
    The memory stack on which the current block is located.

9.4 Operation Table

The operation table consists of the executable instructions, which have the following format.

1. Opcode
   Operation code for the instruction.

2. Op_array1
   First operand array of the instruction.

3. Op_array2
   Second operand array of the instruction.

4. Result_array
   Result array of the instruction.

5. Ind1
   The flowing mx_array_index words of the instruction contain the array indices of the block of the result_array that this instruction references.

6. User_sub
   Pointer into the User_sub table (used if this instruction is an execute).

7. Instr_timer
   Used to carry the timer index if a timer has been registered for this instruction (RT)

8. Op_blkndx
   Pointer to the block manager's data structures indicating the block created as the result of this instruction. This is set when the instruction is executed, and used later in freeing the block at the end of the loop in which it came into scope. (RT)

9. Opblock
   Used to carry the block number (relative to the blocks in result_array) of the block created by this instruction. Used similarly to the Op_blkndx field. (RT)

10. Oploop
    Loop initialization flag. (RT)

11. Pardo_chunk_size
    Used in load-balancing pardo instruction (RT)

12. Pardo_batch_end
    Used in load-balancing pardo instruction (RT)

13. Pardo_next_batch_start
    Used in load-balancing pardo instruction (RT)

14. Pardo_batch
    Current value is "batch" of data being processed by this instruction (if it is a Pardo) (RT).

15. Pardo_max_batch
    Maximum batch that this pardo will process (RT).

16. Pardo_signal
    Used to reset a Pardo instruction (RT).

17. Server_stat_key
    A key passed to the server indicating which line of SIAL code generated this instruction, if it was a REQUEST or PREPARE. Used to track server timing data. (RT)

18. Lineno
    SIAL line number of this instruction.

9.5 Scalar Table

The scalar table simply consists of the values of each of the scalars used in the SIAL program. For instance, if there is a SIAL instruction
Esum = 0.0
then the instruction will have an array table reference pointing to the scalar table. The scalar_table entry pointed to will contain the double precision 0. Some of the scalar_table entries (i. e. those which are linked to pre-defined constants, such as scfeneg and totenerg) are filled in at run-time from parameter data and/or JOBARC data once the actual values are known.