SIAL Programmer Guide

Overview

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 Appendix 1. Example programs are given in Appendix 3.

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, typically 10,000 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 standalone 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.

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.

Memory management

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.

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.

Execution Management

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. Each task knows which blocks, i.e. which block index values, are local and will only loop over those local values.

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 they 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.

Appendix 1: Super Instruction Assembly Language Definition

General requirements:

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

Predefined constants:

Some names are defined from other input sources, such as the JOBARC file. All these constants count in segments, not in individual orbitals.

- **Universal constants**
    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.
- **MO 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 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)
7. **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.
8. **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.
9. **static cb(mu,pb)**: Restricted spin orbital transformation matrix from the SCF, read in from JOBARC. **pb** is mobindex 1:norb and **mu** is aoindex 1:norb.
10. **static e(p)**: Restricted spin orbital energies from the SCF, read in from JOBARC. **p** is moindex 1:norb.
11. **static ea(pa)**: Alpha spin orbital energies matrix from the SCF, read in from JOBARC. **pa** is moaindex 1:norb.
12. **static eb(pb)**: Restricted spin orbital energies matrix from the SCF, read in from JOBARC. **pb** is mobindex 1:norb.

- **Ordering table of predefined constants**: The following tests return true:
  1. `i <= norb`

     where **i** is any predefined constant

  2. `i >= 1`

     where **i** is **bocc, baocc, bbocc, eocc, eaocc, ebocc, bvirt, bavirt, bbvirt, evirt, eavirt, ebvirt**

  3. `i >= 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 >= bocc`
     - `bvirt > eocc`
     - `bvirt > bocc`
     - `evirt >= bvirt`
     - `evirt > eocc`
     - `evirt > bocc`
  5. The 6=4x3/2 relations between the 4 "alpha MO" constants are
     - `eaocc >= baocc`

- `bavirt > eaocc`
- `bavirt > baocc`
- `eavirt >= bavirt`
- `eavirt > eaocc`
- `eavirt > baocc`

6. The 6 relations between the 4 "beta MO" constants are
   - `ebocc >= bbocc`
   - `bbvirt > ebocc`
   - `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.

Predefined special instructions:

Special instructions are defined in the SIP execution environment so that the language does not have to change to add new instructions, which are calls to special subroutines. Available instructions are listed in Appendix 2. Some are listed here with some example use.

1. energy_denominator v(a,i,b,j)
   In coupled-cluster calculations it is necessary to divide a quantity, say Told(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 Told(a,i,b,j) is divided by [e(i)+e(j)-e(a)-e(b)] and the result put into the distributed array T2new(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
```

2. blocks_to_list X
   Write the blocks of array X to a list file. This makes the data available in succeeding SIAL programs. Note: The current implementation of this does not write an actual ACES II list file. 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.

3. list_to_blocks X
   Read the blocks of array X from a list file. The data must have been written by a "blocks_to_list" execution in a preceding SIAL program. Note: The current implementation of this does not read an actual ACES II list file. 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.

4. fmult v(a,i,b,j)

   The instruction fmult is very much like the instruction energy_denominator in that the elements within an array block are effectively scaled. In the case of fmult the instruction

```
execute fmult v(a,i,b,j)
```
multiplies each element of the array block v(a,i,b,j) by the quantity [e(i)+e(j)-e(a)-e(b)] where in coupled-cluster applications the quantities e(i) are the orbital eigenvalues coming from a Hartree-Fock calculation.

Example:
In the following example the distributed array v1(a,i,b,j) is copied into the temporary array temp(a,i,b,j). Each element of the array temp(a,i,b,j) is scaled by [e(i)+e(j)-e(a)-e(b)] and the result put in the distributed array v2(a,i,b,j). The array V1(a,i,b,j) has not been altered (even locally).

```
PARDO a, i, b, j
      GET V1(a,i,b,j)
      temp(a,i,b,j) = V1(a,i,b,j)
      execute fmult temp(a,i,b,j)
      PUT v2(a,i,b,j) = temp(a,i,b,j)
ENDPARDO a, i, b, j
```

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=bavirt,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. `scalar fac`

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

8. `static c(mu,p)`

defines an array stored locally in the task allocated with a separate malloc. All predefined arrays are of this kind.

9. `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.

10. `local c(mu,p)`

defines an array stored locally in the task and allocated on the block stack.

11. `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.

12. `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 distributed and values will be delivered by a server on request. The task number of the server is not specified in the language, but by the environment and is subject to optimization. The SI processor has the option, not controlled by the SIAL program but by the parallel environment directives, to deliver the integrals in one of the following ways:

- compute the block in the calling task,
- request the block from an integral worker task that will then compute and deliver it,
- request the block from an IO manager, that will deliver it from distributed RAM,
- request the block from an IO manager, that will deliver it after reading distributed files on local disks,
- request the block from an IO manager, that will deliver it after reading a parallel file on a global parallel file system
- request the block from an IO manager, that will deliver after obtaining it with GETREC, PUTREC from a serial ACES II style file stored on some local disk or some global disk.

The declaration associates the stated names of the array and the indices with the 2-electron integral matrices that are currently supported. The SI processor decides from the combination of type of indices supplied (**aoindex** or **moindex**) and their

ranges which set of 2-electron indices are meant: AO integrals or partially of completely transformed MO integrals. Currently these are the only arrays that can be declared **served**. 1-electron integrals for hamlitonian and for properties may be added to this list at a later time. Because **served** arrays can overflow to disk, they can be larger than **distributed** arrays.

13. `temp v2(p<q,mu,nu)`

specifies that the indices **p** and **q** are symmetric and that packed storage is allowed; this syntax is allowed on all declarations.

Control statements:

1. **Program**
   - `sial myprog`

     start of an SIAL program called **myprog**. With this control line theprogram can be embedded in any file and all text preceeding 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, if anthing they are like internal procedures in Fortran 90. 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.

   - `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 mu`

     makes control in the loop jump to the statement after the **end do** with the matching variable named on the **exit** statement; a **exit** statement without a variable name generates an assembly error.

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.

Operation statements:

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 widlcard "*" allocates blocks for all values of the matcing 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. **array reference indexing** 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.

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.

9. `v3(p,q,r,s) = x(p,q) ^ 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.

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.

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.

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.

15. `get v3(p,q,r,s)`

    gets the indicated block of a **distributed** array **v3** from the owner task. The shape and sizes of the blocks must match.

16. `prepare v4(p,q,r,s) = v2(p,p,r,s)`

    deliver a block of **v2** to the server to replace the block of the **served** array **v4** for future requests.

17. `prepare v4(p,q,r,s) += v2(p,p,r,s)`

    deliver a block of **v2** to the server to be added to the block of **served** array **v4** for future requests.

18. `request v(mu,nu,lambda,sigma) sigma`

    for **served** array 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**

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 Appendix 2 for a complete list.

22. `execute trace_on|off`

    turn on or off tracing features listed by their keyword.

Appendix 2: List of special instructions

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

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. **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 ststic arrays.

3. **copy_ab**

syntax: execute copy_ab array1 array2
function: The array1 is assumed to be of type alpha-alpha (VaD) or (SD). The elements of the array1 are put into array2 with the FIRST index offset by the number of singly occupied orbitals. Array2 is of type beta-beta (VbD)
restrictions: array1 and array2 must be two-dimensional static arrays. It is assumed that n_alpha > n_beta as is the case in the ACES II program.

4. **copy_ba**

syntax: execute copy_ba array1 array2
function: The array1, whichmust be of type (beta,beta) spin, is copied into array2, which must be of type alpha, with an offset of nalpha_occupied - nbeta_occupied
restrictions: array1 and array2 must be two-dimensional static arrays. It is assumed that n_alpha > n_beta as is the case in the ACES II program.

5. **fmult**

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

6. **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 indeces.

7. 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 segement sizes of 1.

8. 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 indeces 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.

9. 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.

10. 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.

11. 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.

12. 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.

13. 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.

14. 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.

15. 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.

16. return_sval

syntax: execute array1 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.

17. place_sval

syntax: execute place_sval array1 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.

18. square_root

syntax: execute square_root scalar1 scalar2
function: scalar1 is raised to the power scalar2. scalar1 = scalar1**scalar2
restrictions: scalar1 and scalar2 must be declared as scalars in the sial program.

19. apply_den2

syntax: execute apply_den2 source target
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.

20. 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.

21. 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).

22. 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.

23. 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.

24. set_flags

syntax: execute set_flags array1
function: Sets the indices of a 3-d array in common block values. Example: The first index is assumed to be the atom, the second is the component index (i. e. x,y, or z), and the 3rd is the center.
restrictions: array1 must be three-dimensional with simple indeces.

25. set_flags2

syntax: execute set_flags2 array1
function: Sets the indices of a 2-d array in common block values. Example: The first index is assumed to be the atom, the second is the component index (i. e. x,y, or z).
restrictions: array1 must be two-dimensional with simple indeces.

26. 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-dimesional array with AO indices and the perturbation must have been set, by set_flags2 probably.

27. 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.

28. 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 betwo-dimensional array with AO indices. The perturbation must have been set before fock_der is called.

29. 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.

30. 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.

31. 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 indeces.

32. removevv_dd

syntax: execute removevv_dd array1 array2
function: removes all doubly occupied indeces from the array1 with array2 being the result of the array with the all doubly occupied indeces 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.

33. removeoo_dd

syntax: execute removeoo_dd array1 array2
function: removes all doubly occupied indeces from the array1 with array2 being the result of the array with the all doubly occupied indeces 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.

34. 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) indeces. a/i -> alpha_virtual/alpha_occupied. Only used for ROHF codes

35. 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

36. 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

37. 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) indeces. b -> beta_virtual OR (i,i), i -> alpha_occupied. Only used for ROHF codes

38. 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.

39. 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.

40. 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 indeces as the fock array.

41. 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

42. print_scalar

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

43. dump_block

syntax: execute dump_block array1(p,q,r,s)
function: Writes out information about the block af array1(p,q,r,s). The first,last,maximim,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.

44. array_copy

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

45. 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

46. blocks_to_list/write_blocks_to_list

syntax: execute blocks_to_list array(p,q,r,s)
syntax: execute write_blocks_to_list
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

47. list_to_blocks/read_list_to_blocks

syntax: execute list_to_blocks array(p,q,r,s)
syntax: execute read_list_to_blocks
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.

Appendix 3: Example Programs

Example 1: 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
beallowed
  # to start so that of all accumulates in progress at least one
  # is ready everytime accumulate is executed by the SIP
  put v4(p,q,r,s) += v4(p,q,r,s)
  return
end proc 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
```

Example 2: 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
```

Example 3: Using served arrays efficiently.

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.