@manual

For serial manual visit http://www.giss.nasa.gov/molscat

CCP6 MOLSCAT

User Manual for parallel implementation, version 1

(Based on version 12 of the serial program)

I.J.Bush

Daresbury Laboratory,
Daresbury, Warrington
WA4 4AD
Cheshire

Electronic mail: I.J.Bush@daresbury.ac.uk

November 12, 1999

Prerequisites: A familiarity with both the serial version user manual and code is assumed. The user manual is available by anonymous ftp from krypton.dur.ac.uk in the file /pub/molscat.v12.doc

Contents

1 Introduction

This document describes a limited parallel implementation of the CCP6 program MOLSCAT, a general purpose package for performing non-reactive molecular scattering calculations. The code has been run successfully on a 64 node iPSC/860 and a small HP 700 series cluster at Daresbury.

The aim of the implementation is three fold:

- To cut down on the large memory requirements of the program by distributing the larger arrays across a number of processes.
- To show reasonable scalability.
- To make the user interface as similar to the serial version as possible.

1.1 Implemented sections of the code

The following portions of the code have been implemented:

- All non-IOS coupling types (but only
  - Atom - linear rigid rotor scattering (ITYPE = 1)
  - Linear rigid rotor - Linear rigid rotor scattering (ITYPE = 3)
  - Atom - asymmetric top scattering (ITYPE = 6)
  - Atom - asymmetric top scattering within the centrifugal sudden approximation (ITYPE = 26)
  - The generalised form of atom - vibrating diatom scattering, allowing a j dependent potential (ITYPE = 7)

  have been tested).
- The diabatic modified log-derivative propagator of Manolopoulos (INTFLG = 6), both with and without propagator scratch files.
- The hybrid modified log-derivative Airy propagator of Manolopoulos and Alexander (INTFLG = 8), both with and without propagator scratch files.
- Checking of the convergence of the S matrix.
- Resonance searching.

1.2 Sections of the code that are *NOT* implemented and planned future developments

Besides the non-implemented propagators, the following features of MOLSCAT are not currently available in the concurrent version:

- IOS Calculations
- Pressure broadening calculations
- The LASTIN option
- Convergence checking of total cross section calculations

Work is currently in progress on implementing the pressure broadening section of the code, and it is also planned to allow the LASTIN option.

2 Input files: Names and data formats

2.1 The main input file

The communications harness about which the code is designed is PVM. This unfortunately places some restrictions on where the input file may be placed in the file system. Further the fact that many processes may attempt to read this file means that simply redirecting standard input will not work. Due to these considerations the file must be in the home directory of the user on the machine on which the process is executing, and it must be called molscat.in. This name may be changed by altering the INPUT_FILE parameter in the driver routine and re-compilation.

2.1.1 The NAMELIST block & INPUT

The interpretation of the variables defined in this block is identical to the serial program, with one very minor exception. If JTOTL>JTOTU and JSTEP<0 then the parallel code will step down from JTOTL to JTOTU. The serial version would accumulate total scattering cross sections until convergence had been reached.

2.1.2 The NAMELIST block & BASIS

The interpretation of the variables defined in this block is identical to the serial program. However users intending to supply their own routines (ITYPE = 9) should examine the sections on how the coupling constant array (VL) is distributed, and should also see the note at the end of this subsection.

2.1.3 The NAMELIST block & POTL

The interpretation of the variables defined in this block is identical to the serial program. However users intending to supply their own routines should see the note at the end of this subsection.

2.1.4 The NAMELIST block & CONV

The interpretation of the variables defined in this block is identical to the serial program.

2.1.5 A note on user supplied routines

User supplied routines should *not* exit the program by using the FORTRAN statement STOP. This will almost certainly cause deadlock. Instead the utility subroutine DIE, which is supplied with the program, should be called.

2.1.6 A Warning about Channel Use for Scratch Files

In the parallel version of MOLSCAT FORTRAN channel number 1 is reserved for internal use. It should not be used for scratch files.

2.2 Other input files

The parallel version may, but need not, use two other input files. One must be called par_control.in and must be situated in the user's home directory, as for molscat.in. This file sets up various options that control the exact manner in which the parallel program executes. However if the user requires non-default options it will be rarely necessary to create this file for it may be created by command line options. The other's name may be set by the user. This file controls a list of all the JTOT/M/Energy values that the job will use. This is discussed latter in the section on job control.

3 Output files: Names and data formats

With the exception of the log file the data formats for the output files are the same as for the serial code. The log file's format is very similar to the serial version, but

- it may contain informational messages describing the data distribution and parallelisation in the job.
- the 'raw', unshifted values for the eigenphase sums are written out. This is because a given process in the parallel code has no guarantee that it will know what the previous eigenphase sum was.
- any calculations with no open channels are skipped before any propagation occurs.

However the names of the files differs from the serial version. Each process writes to its own set of files, called <old name>.xxxx, where xxxx is a number unique to each process, ranging from 0 to one less than the number of processes in the job. The log file is called molscat.out.xxxx. Thus a three process job may create the following files:

molscat.out.0000	molscat.out.0001	molscat.out.0002
ISAVEU.0000	ISAVEU.0001	ISAVEU.0002
ISIGU.0000	ISIGU.0001	ISIGU.0002
KSAVE.0000	KSAVE.0001	KSAVE.0002

etc.

The potentially large number of open files could conceivably cause problems on some machines. In latter versions a different solution to the problem of output may be desirable.

4 Running parallel MOLSCAT

4.1 Basic execution

The exact methodology for executing the parallel version may, and probably will, vary slightly from machine to machine. The following description assumes that a workstation cluster of machines running under UNIX is the platform.

1. Check that a version of PVM 3.2.1 or later is installed on all the machines.
2. Start the PVM daemon on the machines, ensuring that the hosts file tells PVM where the daemon and molscat executables reside.
3. In the file molscat.script check that the file assigned to the variable host_executable is the host program.
4. Execute molscat.script

You will then be prompted for the name of the executable and how many processes will be used in the calculation, and then the calculation will start.

4.2 Command line options

It will probably be sensible to put molscat.script somewhere that is on your PATH and then set up an alias e.g.
in csh
alias molscat "molscat.script $argv"
in ksh
molscat(){
molscat.script $@
}

Then the full syntax of the molscat command becomes:

molscat[-he executable] [-we executable] [-nod #] [-o file]
[-inquire] [-all] [-min #] [-[no]sort] [-read file] [-help]
where -he executable Specifies the host program executable.
-we executable Specifies the worker program executable

-nod #	Specifies the number of processes in the concurrent job. Default is 2.
-o file	Specifies the log file for the host program.
-inquire	Perform an inquiry run. This performs a run that quickly ascertains how many processes are required for the given job.
-all	Perform job control in 'all energies' mode (see section on job control).
-min #	Always distribute VL over at least # processes
-[no]sort	[Don't] sort the job control list. Default is -sort. (see section on job control).
-read file	Read job control list from file. (see section on job control).
-help	Lists a summary of the available command line options. If -help is specified then no calculation is performed.

All these options are actually to set up the par_control.in file mentioned in section 2.2, and thus the user will rarely, if ever, have to actually create this file.

5 General overview of the Parallel package

The package consists of three different executables, each of which have a different task to perform. These are the shell script molscat.script, as mentioned above, a host program and a worker program.

5.1 The shell script

The shell script is written in standard Bourne shell (sh) commands, and should therefore be portable across many Unix machines. It is fairly simplistic and may well not do exactly what you think it should do if you make mistakes on the command line. Expert shell programmers are invited to write a better one !

5.2 The host program

The host program is written in fairly standard FORTRAN 77. The extensions it uses, as flagged by the HP f77 compiler with the -a option, are

- Variable names longer than six alphanumeric characters.
- Variable names containing the underscore (_) character.
- Include files are used.
- Lower case is used.
- A routine to flush FORTRAN channel 6 is used.

However these are extremely common, if not universal, extensions to FORTRAN 77, and the first four are acceptable under the Fortran 90 standard. The last extension is not vital to the execution of the program, and in the absence of such a routine the executable statements in STDOUT_FLUSH may be commented out. It also has to be linked with the PVM libraries.

The purpose of the host program is to decide what task the worker processes will next perform. At the start of the job it spawns the required number of workers and then waits until it receives a message from one of the them. This message contains a list of all the jobs to be performed, their sizes and how many processes will be required to perform each one. It again waits until one process sends it a message telling it that it is available for a job. If the first job requires only one process the host will send off the job description to the process which signalled it, otherwise it will wait until enough processes are available for work, and it will send out the job to them. It will repeat this process until all the jobs have been performed, and then it will signal to the workers that the calculation is complete and exit.

5.3 The worker programs

The worker processes actually do the calculation, and resemble in most places the serial version of the code. The extensions to standard FORTRAN 77 are those mentioned in the serial code's user guide and

- Variable names longer than six alphanumeric characters.
- Variable names containing the underscore (_) character.
- Include files are used.
- Lower case is used.
- A routine to flush FORTRAN channel 6 is used.

Similar comments to those on the host program apply.

After being spawned by the host, the worker reads molscat.in, processes the data and initialises the various data sets required by the main calculation. Then for every JTOT/M/Energy combination the number of basis functions are calculated, and from that the number of processes required to hold the VL array for that combination. Note that this is for *every* JTOT/M/Energy combination, thus the counting calls to BASE are done before the main calculation loop is ever reached, unlike the serial code. This list is then processed into a form to be sent back to the host. This consists of a list of jobs, each job description consisting of the following data:

1. JTOT
2. M
3. Number of basis functions
4. Number of processes over which VL will have to be distributed

5. E(1), E(2), E(3) ... E(processes) the energies for this job.

They then send this list back to the host, finish initialisation of the job, and then signal to the host that they are ready for work. They wait until the host sends them a job description of the form above. Each process which receives this job description then decides on which energy it will propagate (each process chooses a different energy) and which parts of the coupling array will be calculated and stored by it. The coupling constants are then calculated and the wavefunction propagated. Each process then outputs the desired results for this propagation to its set of files, and then signals the host that it is again available for work.

If the host signals to the process that there is no more work to be done any final output and analysis is performed, and the worker then exits.

6 Groups

Before giving a more detailed description of both the host and worker programs it is necessary to introduce the concept of groups as applied to MOLSCAT. A group is simply a number of processes working together. In the parallel implementation of MOLSCAT there is a three-tier hierarchy of groups, each group a subset (or identical) to the group above it in the hierarchy. The groups from the largest to the smallest, are:

- The global group, g_g. This group contains all processes in the job except the host program.
- The working group, g_w. This group contains all processes working together on the same JTOT/M combination.
- The VL group, g_V. This group contains a set of processes over which VL is distributed.

Thus in MOLSCAT a group simply has two properties, a name and a size, n_g. Within a given group each processes has two properties:

- An instance number, i_g. This is a unique identification number for each process, $0 \leq i_g < n_g$.
- A connectivity to other processes in the group. In MOLSCAT this is set up as a ring, processes i_g being connected to processes $\mathrm{mod}(i_g - 1, n_g)$ and $\mathrm{mod}(i_g + 1, n_g)$

For instance in a group of size 4 the processes have instance numbers 0, 1, 2, and 3, process 0 is connected to processes 1 and 3, process 1 to 0 and 2, 2 to 1 and 3, and 3 to 2 and 0. It should be noted that the i_g of a process in one group will generally be different from its instance number in another group. Further the difference between the identification number of a process and its various instance numbers should be appreciated. The identification number of a process is a unique number ascribed to each process *by the message passing harness*. They are thus harness dependent, may well vary from run to run, and

could well be machine dependent. The instance numbers are assigned *by the program*, and are totally invariant to such considerations.

As a slight aside the instance numbers in the global group have already appeared. They are used as the postfixes on the output file names.

There are many routines supplied to utilise the three groups, thus making the communication harness in most cases transparent to the user. These routines are documented in section A of the appendix, along with any other communication routines.

7 Job Control

Each process in a working group will propagate the wavefunction under slightly different conditions. For each process JTOT and M will be the same, and so the basis functions and hence the coupling constants will be the same for each process, but typically each process in the group will propagate for a different energy. The one exception to this is in convergence checking, when the different processes use slightly different integration parameters.

However it will not always be possible to find useful work for every process. Imagine the situation where VL is so large that it requires four processes to store it, but propagation is required for only three energies. The VL group must contain at least four processes, and so one process will be redundant for propagation purposes. In such a case all output, except for severe warnings and fatal errors, is switched of for that process, and it simply supplies the relevant parts of VL when required. I shall term such a process as being in 'VL server' mode.

Obviously the optimal job control method for a particular calculation will:

- Minimize the number of processes acting as VL servers
- Minimize the time spent by a process waiting for more work from the host
- Minimize the CPU and wall time required.

These goals are extremely difficult to meet without resorting to a highly complex control system which would have to calculate how long each job will take, and then find the optimal order to send the jobs out to the workers. Instead two different simple job control systems are available, 'default' and 'all energies,' and it is also possible to define the exact sequence in which the jobs will be performed by hand.

7.1 Default Job control

'Default' job control is actually only the default if the program is not searching for resonances, convergence checking, or the job is being run on one process ('pseudo-serial'). In this case the working and VL groups are identical. Each process propagates for one energy (or acts as a VL server) and then requests more work from the host. This continues until all the jobs are completed. As an example consider the case when JTOT=10, M=1,2, there are four energies, four

processes and each calculation requires VL to be distributed over two processes. The job list that the host will receive will look something like

JTOT	M	N	Processes	Energies
10	1	300	2	1 2
10	1	300	2	3 4
10	2	332	2	1 2
10	2	332	2	3 4

The first two processes to signal the host, say their instance numbers in the global group are 1 and 2, will receive the job JTOT=10, M=1, energy=1,2. Process 1 will do energy=1, while process two will do energy=2. Next processes 0 and 3 will signal the host and receive the second job in the list. Since the higher the energy the longer the propagation tends to take, processes 1 and 2 will finish first, signal the host that they are ready for more work, and receive the third job in the list. Here the inefficiency in the scheme becomes apparent. Processes 0 and 3 receive two comparatively compute intensive jobs, while 1 and 2 get off comparatively lightly, and it may well have been more efficient to have each group do one short and one long job.

An even worse example of inefficiency would have been if the job control list was

JTOT	M	N	Processes	Energies
10	1	300	2	1 2
10	1	300	2	3 4
10	2	332	3	1 2 3
10	2	332	3	4 0 0

(The zeros in the list indicate VL servers). In this case, as before processes 1 and 2 would do the first job, while 0 and 3 would do the second. However when processes one and two finish they must wait for either 0 or 3 to indicate that they are available for work. Say 0 finishes first, processes 0, 1, and 2 do the third job. Meanwhile process 3 finishes and indicates to the host that it is ready for a job. However it must wait for two of the other processes to finish before it can proceed. When two of them do, process 3 and say 0 and 1 will then do the fourth job. But for this fourth job the VL matrix is the same as for the third, but process 3 does not have this set of coupling constants stored on it. Therefore a potentially expensive call to BASE has to be made, which could have been avoided if the job allocator was intelligent enough to work out that it would have been better to use processes 0, 1 and 2 for both the third and fourth jobs. It is partially to circumvent this problem that the 'all energies' job control mode is available.

7.2 All Energies Job Control

In the 'all energies' mode of job control one working group propagates all the energies for a given JTOT/M combination. This is the default mode for reso-

nance searching, convergence checking and for 'pseudo-serial' calculations, and it can be forced by the -all command line option.

In this case the VL group will usually be smaller than the working group. Consider the first case above. In all energies mode the job allocator will realise that if a working group of four is created, and two VL groups each of two processes are created within that group, all four energies may be done. For this case the job control list is

JTOT	M	N	Processes	Energies
10	1	300	4	4 0 0 0
10	2	332	4	4 0 0 0

The energy list in all energies mode is redundant, for obvious reasons. In this case there is no particular advantage in using this mode. However in the second case the job list becomes

JTOT	M	N	Processes	Energies
10	1	300	4	4 0 0 0
10	2	332	4	4 0 0 0

The way that groups are created ensures that the same set of 3 processes do all four energies when M=2. This is still not ideal, but a call to BASE has been avoided.

This form of job control should, however, be used with caution. Say in the first case above there were not 4 but 6 processes in the job. The first four processes to signal the host would be sent the JTOT=10, M=1 job. The next two would signal the host, but the host thinks that at least four processes are required for the second job, and so will wait until two of the other processes are available. Compare this with the default action. The first two processes will get the first job, the next two the second and the last two to signal the host the third. This leads obviously to a more efficient use of the resources.

7.3 Manual Control of the Job Control List

Both the two automatic forms of job control have, as been seen above, potentially serious drawbacks. Potentially the way to get the best performance is to read in a job control list using the -read command line option, and probably with the -nosort option as well.

In the job control file every possible combination of JTOT, M and energy should be listed in the format
JTOT M Energy N Required number of processors.
Thus to reproduce the job control list for the first example the job control file would contain
10 1 1 300 2
10 1 2 300 2
10 1 3 300 2
10 1 4 300 2

10 2 1 332 2
10 2 2 332 2
10 2 3 332 2
10 2 4 332 2
Using this method one can improve the performance of the second example by using the job control file
10 1 1 300 2
10 1 2 300 2
10 1 3 300 2
10 1 4 300 2
10 2 1 332 4
10 2 2 332 4
10 2 3 332 4
10 2 4 332 4,
that is forcing slight over parallelisation on the M=2 jobs by distributing VL over 4 processes, not just three. Therefore only one pass, not two over the M=2 energies need be performed.

Unless the -nosort option is set on the command line the job control list is sorted. Usually it is sorted in increasing job size, as measured by the number of basis functions, but if JSTEP<0 is set in molscat.in this ordering is reversed. If two jobs are the same size the one with the lower JTOT comes first in the list.

The main problem with this approach is that it requires a much greater understanding of how the program works to be able to use it effectively. However the judicious use of the -inquire option, which causes the program to output the job control list and then exit, is a straightforward way of tailoring the list to be as efficient as possible.

7.4 The Job Control Code

A brief description of the major subprograms involved in job control is included in section B of the appendix.

8 Memory Management

As in the serial version the large array X is used to provide effective dynamic memory allocation, and the organisation within it is very similar in the parallel version. The differences result from

1. Distributing VL
2. Further space saving tricks, mainly to do with the real and imaginary parts of the S matrix.

8.1 The Distribution of VL

VL, the coupling constant matrix, is a three dimensional hypermatrix, its elements normally subscripted $VL_{ij\lambda}$ where $i,j \leq n$, the number of basis functions,

and lambda $\leq$npotl, the number of terms in the potential. It is symmetric in the i and j indices, and advantage is taken of this in both the serial and parallel versions of the program to save on storage space.

For the parallel version where VL is distributed it is convenient to think of the array as n (two dimensional) rectangular matrices $\mathbf{VL_i}$ of dimension npotl by i, i$\leq$n. The aim is to distribute these matrices across the processes in the VL group in such a way that the storage taken up on each process by its part of VL is, as close as possible, the same. One of the simplest ways of achieving this is as follows. Consider a very simple and unrealistic example where n is four and it is intended to distribute VL across two processes. Thus the set of matrices, and the storage each one takes up, is

$\mathbf{VL_1}$ requires npotl storage units
$\mathbf{VL_2}$ requires 2$\times$npotl storage units
$\mathbf{VL_3}$ requires 3$\times$npotl storage units
$\mathbf{VL_4}$ requires 4$\times$npotl storage units.

If the process with instance number 0 is assigned matrices 1 and 4, while process 1 gets matrices 2 and 3, each process will store exactly the same amount, 5$\times$npotl. This idea can obviously be extended to the general case. Process 0 gets a few of the very smallest and very largest matrices, Process 1 is assigned a set of slightly larger small matrices and slightly smaller large matrices and so on. More specifically let i_g be the instance number of the process under consideration, n_g be the number of processes in the VL group, $n_{av} = \mathrm{int}(\frac{n}{n_g})$ and $n_s = \mathrm{int}(\frac{n_{av}}{2})$. Then this process will be assigned the matrices $\mathbf{VL_i}$ for $n_s \times i_g + 1 \leq i < n_s \times (i_g + 1)$ and $n - n_s \times (i_g + 1) + 1 \leq i \leq n - n_s \times i_g$. This will only cover all the matrices if the number of basis functions is an even multiple of the number of processes in the VL group. If this condition is not satisfied the small number of matrices not covered by the above recipe are assigned in order to different processes. The resulting imbalance is negligible.

8.1.1 The Routines which Control the Distribution of VL

This section provides a general overview of the routines concerned with distributing VL across the processes. For more syntactical detail see section C of the appendix.

There are four routines concerned with the distribution of VL. Three are simple integer functions

- VLnods returns the minimum number of processes over which VL must be distributed to fit in the large X array
- VLmem returns essentially the memory that VL will use in the X array
- VLnumi returns how many of the (2D) matrices this process will hold.

However the main workhorse of the VL distribution system is the subroutine VLlist. This, through its arguments, returns three pieces of important data:

- integer number_of_i. This is the number of (2D) matrices this process will hold, i.e. it is the same as the return value of VLnumi. It is commonly abbreviated in the code to NUMI.

- integer matrices_this_node(1:number_of_i). This holds a list of the (2D) matrices held by this process, indexed by I value. It is commonly abbreviated MATNOD.

- logical held_by_this_node(1:n). If the (2D) matrix $\mathbf{VL_i}$ is held by this process, HELD_BY_THIS_NODE(I) is true. It is commonly abbreviated HLDNOD.

The two arrays are held in the large dynamic memory (X) array, and space must be allocated for them.

In general VLlist can be treated as a 'black-box' routine, though its methodology is a straightforward implementation of the ideas outlined above. However it has a secondary purpose, to generate a list of the communications involved in the propagators. For more detail see the later. In passing it should be mentioned that VLlist is *always* called, even when VL is to be all held on one process. In such a case

- number_of_i = n

- the elements of matrices_this_node are 1, 2, ... n

- all elements of held_by_this_node are set to true.

Though described as the workhorse, VLlist is not a computationally expensive routine, being $\mathcal{O}$(n) instructions.

8.1.2 User Implemented Coupling Schemes

After a call to VLlist has been made the implementation of most user defined coupling schemes is very straightforward. Simply change the lines corresponding to

```
      do xx i = 1, n
```

to

```
      do xx index = 1, number_of_i
         i = matrices_this_node( index )
```

where xx is a line number. However many schemes count the number of zero elements of the VL hypermatrix, and report a warning if all elements are zero. In the parallel version of MOLSCAT all elements of VL on one process may be zero, but this does not imply that *all* elements of VL are zero. The simplest way to avoid spurious warnings is to change the code to something like

```
.
.
      integer     work
      integer     dummy
      character*7 group
      character*7 VL_group
.
.
      dummy = 0
.
.
      group = VL_group( dummy )
.
.
      number_not_zero = 0
      do xx index = 1, number_of_i
         i = matrices_this_node( index )
.
.
         do yy j = 1, i
.
.           Calculate relevant elements of VL
.
            if( element .ne. 0.0d0 ) then
               number_not_zero = number_not_zero + 1
            endif
.
.
yy       continue
xx    continue
```

```
.
.
      call group_imax( number_not_zero, 1, work, group )
      if( number_not_zero .eq. 0 ) then
         write( 6, xxxx )
      endif
.
.
 xxxx format( 1x, 'All elements of VL are zero !')
```

where the routines VL_GROUP and GROUP_IMAX are described in the appendix. It is *very* important to note that the fragment

```
.
.
      if( number_not_zero .eq. 0 ) then
         call group_imax( number_not_zero, 1, work, group )
      endif
.
.
```

is **INCORRECT**. For GROUP_IMAX not to result in deadlock *all* processes in the relevant group *must* call it. In the above fragment it is perfectly possible that only a small number of members of the group will call it.

There is one other small change that may be required. On a basis function counting call to BASE the J array must be calculated. This will require storage in the dynamic memory array.

8.2 Other Changes to the Memory Management

By far the most important change not to do with VL is the handling of the real and imaginary parts of the S matrix. These are symmetric matrices, and are now stored in packed upper format i.e. the order of storage is

S(1) in the program corresponds to S(1,1)
S(2) in the program corresponds to S(2,1)
S(3) in the program corresponds to S(2,2)
S(4) in the program corresponds to S(3,1)
S(5) in the program corresponds to S(3,2)

etc. This means the SR and SI need only be allocated $\frac{n(n+1)}{2}$ storage units in the dynamic memory array (X), a saving of n(n-1) over the serial code. There are some important consequences of this:

- Much of the code generating and handling the S matrices has been completely rewritten.
- The Y and W arrays are also held in packed upper format.

- The memory management in some places has become more complex.

The third of the above points requires more elucidation, particularly with regard to two points. Firstly though the W matrix is held in packed format it must be allocated n^2 storage units. The reason for this is two fold. Firstly in many places a n^2 scratch array is required. The area in X corresponding to W is often used for this. Secondly the second half of the W array is sometimes used as temporary buffering for communications.

The second point requiring elucidation is the interface to the YTOK routine, the subprogram that converts the log derivative matrix to the K matrix. Here, in essence, two scratch matrices of size n^2 are required, while, without taking up further space in the dynamic memory array, only one is available. Here advantage is taken of the fact that SR and SI are stored contiguously in the X array, and the arguments corresponding to SR and SI for YTOK (Y and T) are next to each other. In the parallel implementation the T array is unused, but the Y array is declared as Y(1:N, 1:N), and thus, though T is not explicitly mentioned in the routine, part of it will be used when accessing the second half of Y.

There are two other small changes in the use of the dynamic memory array:

- While it is being constructed the job list is placed in the array. After it has been sent to the host it is deleted.
- A basis function counting call to BASE now allocates the J array. Space on X must be available for this, as noted above.

9 Propagating the Wavefunction

Virtually all the communication involved with propagating the wavefunction occurs in the routine WAVMAT, which has been completely rewritten. The next subsection gives a description of the methodology employed, and that following it outlines some other consequences of this.

9.1 The Routine WAVMAT

The heart of the two propagators implemented in the parallel version of MOLSCAT is the routine WAVMAT. This oversees the calculation of the (symmetric) W matrix from

$$W_{ij}(r) = \sum_{\lambda=1}^{npotl} VL_{ij\lambda} p_\lambda(r)$$

where **p** is the vector representing the potential for a radial separation r. (This is not exactly the case, there are some additional terms in the diagonal elements of **W**, but since **W** is not distributed these may be calculated locally, exactly like the serial code.)

The method used is to pass the potentials calculated on each process in the VL group round the ring and send the parts of **W** back to the process

from which the potential originated. More specifically, if $\mathbf{p_n}(r)$ is the vector describing potential calculated on process with instance number n in the VL group, and the size of the VL group is n_g, then

1. For process with instance number n calculate $\mathbf{p_n}(r)$
2. Let COUNT=1
3. Let ORIGIN=mod(n-COUNT+1, n_g)
4. Let NEW=mod(n-COUNT, n_g)
5. Send $\mathbf{p_{ORIGIN}}(r)$ to instance number mod(n+1,n_g)
6. Receive $\mathbf{p_{NEW}}(r)$ from instance number mod(n-1,n_g)
7. $\forall \mathbf{VL_i}$ on process n calculate $\mathbf{W_i}(r) = \mathbf{VL_i p_{NEW}}(r)$
8. If COUNT $< n_g$ then
   (a) Send $\{\mathbf{W_i}\}$ back to instance number NEW
   (b) Receive message from mod(n+COUNT,n_g) containing the parts of **W** calculated on that process
   (c) Fill in the parts of **W** represented by this message
9. COUNT=COUNT+1
10. If COUNT $< n_g$ return to step 3

This looks a lot more complicated than it actually is ! As an example let us focus on the process with instance number 0 in a 3 process VL group. The first thing it does is calculate the potential vector. It then initializes a variable COUNT to 1, and sends the potential onto the next process in the ring, instance number 1. It then receives the potential from the previous process in the ring, number 3 and uses that to calculate all the parts of **W** that it can with its part of **VL**. From COUNT the process knows that only 1 potential send has been made, so the potential used in the calculation of **W** must have come from process 2. Hence the part of **W** just calculated belongs on that process, and so it is sent there. Similarly because COUNT is one the potential calculated by process 0 is at present on process 1, and so process 1 has just calculated part of the **W** matrix that should be on process 0. Therefore node 0 receives it. Further because the **VL** hypermatrix is distributed in a completely deterministic manner node 0 knows not only what parts of **VL** are held by it, but also what parts are held by all other processes in the VL group. It follows from this that it can work out which parts of the **W** matrix the message it has just received from process 1 correspond to, and so the **W** matrix can be partially constructed. COUNT is then incremented, becoming two, the potential on process 0, which originated on process 2, is sent to process 1, and a new potential is received from process 2. Again the parts of **W** that can be calculated are done so, and from the value of COUNT it knows that this belongs

on process 1, from where the potential originated. Similarly it knows that the part of **W** relevant to it is at present on process 2. These are received, and since each process holds a different part of **VL**, a different part of **W** can be filled in. COUNT is again incremented, the present potential is sent to process 1, and a new potential is received from process 2. Now COUNT is three, and therefore the potential on process 0 is the potential that was originally calculated on that process. Hence the part of **W** that will be calculated is the part required on this process, and therefore all calculations can be done locally. A similar sequence of events occurs on the other processes.

The actual implementation is an optimised version of this. The two main features that are not outlined above are

- All the communications occur asynchronously, thus reducing the time the process sits waiting for messages.
- All the details of the communications are precomputed in VLLIST and are communicated to WAVMAT through common blocks to hide the details from the user.

For more specific details see the code which is well documented.

This method makes no assumptions about the potential beyond the vector having the same dimension on each process. In general the potential will actually be the same on each process and the potential communications will be redundant. However they are short and asynchronous, and so incur only a very small drop in performance, and the flexibility gained for cases when the potentials are *not* all the same, for instance in convergence checking, more than outweighs this.

9.2 Implications of the Implementation of WAVMAT

The passing of messages around the ring means that all processes must call WAVMAT before the calculation of **W** on any one process can complete. Thus WAVMAT is a 'loosely synchronous' routine. This has the important implication that each member of the VL group *must call WAVMAT exactly the same number of times.* This problem must be addressed in two different circumstances.

In DASCAT (the routine corresponding to propagator 6) the number of propagation steps is calculated before the main loop. In this case each process calculates how many steps it would use if it where performing the propagation independently. Then a GROUP_IMAX call is used to choose the largest number of steps wanted, and the step size is adjusted accordingly.

In AIRPRP (the routine corresponding to propagator 8), and in the routines used to find a suitable lower bound for the separation to start propagating from (FINDRM and RMSET), rather than there being a predetermined number of steps, calls to WAVMAT are made until a convergence criterion is satisfied. In this case a logical variable is used which becomes true when the criterion is satisfied on the process, and then a GROUP_LAND is used to determine if all processes have reached convergence. If they have the routine is

finished, if not it carries on calling WAVMAT until the GROUP_LAND returns true.

10 Resonance Searching

The implementation of resonance broadening is very straightforward. Once five energies have been propagated (5 points are required for the interpolation method) the eigenphase sums are broadcast to all processes in the working group, shifted and the resonance is found as in the serial code. Note that all processes at this point will be doing exactly the same thing, an inefficiency, but as the routine which searches for resonances is quick this is not of importance.

Users of the serial version should note that the main driver routine does not call NEXTE, the routine which searches for resonances, in the parallel version. Instead it calls RESONANCE_SEARCH which oversees the communication of the eigenphase sums and their shifting before it calls NEXTE.

11 Convergence Checking

The method used to implement convergence checking is simply to pass the S matrices around the working group ring, comparing the one just calculated with the one on the previous process in the group. The ordering of jobs within the group is designed so that the correct pairs of matrices are compared. On the first call to the convergence checking routine process 0 may read S matrices from a file, on later calls it uses the S matrices from the last process in the group calculated on the last call.

Some clarification may be necessary with regard to the case where the number of steps in the propagator is varied. Previously it was stated that all processes must call WAVMAT the same number of times, and this may appear to be in conflict with such a convergence check. However as that convergence checking is run in 'all energies' mode, the VL and working groups will be different. The statement about calling WAVMAT refers to the VL group. The convergence checking occurs within the working group. There is therefore no conflict.

12 Other Changes to the Program

There have been a large number of other small changes to the program, too numerous to list here. Generally they consist of either one or two extra arguments in an argument list of a subprogram, or the removal of a small inefficiency, or just some tiding up of some slightly obscure code. Users used to the serial code should check these, especially the argument lists, before modifying the code.

13 Implementation Dependent Parameters in the Include Files

Some parameters in the include file may have to be changed depending on the actual machine that the package is being installed on. The important ones are

- MAXNOD in comm.finc sets the maximum number of worker processes that may be used.
- VL_MEGABYTES in max_VLmem.finc sets the maximum number of megabytes VL may occupy on one process.
- The parameters in size.finc set the sizes, in bytes, of various variable types. They are set up for 32 bit machines, and will require modification if the package is to be used on a 64 bit machine.

14 Some Sample Performance Figures

Benchmarking is a tricky business, fraught with dangers. It is bad enough in a serial environment, but in parallel the situation can be much worse. Therefore the following figures should be taken with an extremely large pinch of salt, and be merely taken as general guidelines rather than gospel truth.

The runs were for propagator number 6, coupling type 6 and 8 energies, all at one JTOT/M combination. There were nine terms in the potential. The size of the job was varied by altering the JMAX parameter. The runs where performed on an HP 700 series cluster, with communications occurring over FDDI, and full optimisation at the compile stage. Default job control was used. In all cases VL was distributed across all processes in the job. Since the load balancing is not perfect the time recorded for the parallel jobs is the time taken by the most compute intensive process, and the memory requirement is that for the most heavily loaded process. Time is in CPU seconds and memory in double precision storage units.

JMAX	N	Procs	Time	Memory	Speed up
5	67	1	43.1	35674	—
5	67	2	34.0	25492	1.3
5	67	4	22.5	20264	1.9
5	67	8	18.1	17808	2.4
7	164	1	486.2	193679	—
7	164	2	301.4	132712	1.6
7	164	4	192.3	102215	2.5
9	325	1	5461.0	730500	—
9	325	2	2261.9	492864	2.1(!)
9	325	4	1392.5	372299	3.4
9	325	8	850.6	312120	5.6

These figures, as far as they go are rather encouraging. There is appreciable speed up and memory savings even for rather low values of N. Further the integrator used is the less compute intensive, and the number of terms in the potential expansion was rather small. On using INTFLG=8 and a more complex potential one would expect some further improvement.

The Appendix

The appendix covers the more important routines and include files introduced into the code on parallelisation.

A Communication Routines and Utilities

In the following:

- integer id is the identification number assigned to a process by the message passing harness (PVM).
- integer instance is the instance number of the process in the given group
- integer dummy is a dummy argument
- character*7 group is the name of a group

A.1 Basic communication routines

These routines are relatively low level, and interface directly with the harness.

subroutine inicom
Initialise the communications.

subroutine exit_comms
Exit the communications harness.

subroutine synchronous_send(msg_tag, type, buffer,
size, id, tag)
integer msg_tag, type, size, id, tag
TYPE buffer(1:size)
Synchronous_send sends a message of size SIZE held in the buffer BUFFER to the process with identification ID. The message is of type TYPE, valid types being defined in the file types.finc. The message is given an identification number MSG_TAG. TAG is redundant in the PVM implementation. The communication is synchronous in the sense that no further computation will occur until the message is safely on its way, but not necessarily received.

subroutine synchronous_receive(msg_tag, type, buffer, size, id, tag)
integer msg_tag, type, size, id, tag
TYPE buffer(1:size)
Synchronous_receive receives a message of size SIZE and puts it in buffer BUFFER from the process with identification ID. The message is of type TYPE, valid types being defined in the file types.finc. The message must have an identification number MSG_TAG. TAG is redundant in the PVM implementation. The communication is synchronous. No further computation will occur until the message is received.

There are also asynchronous versions of the above routines. For further information see the code.

integer function who_am_i(dummy)
This function returns the identification number of the process from which it is called.

integer function size_of_job(dummy)
This function returns the total number of processes in the job, including the host.

A.2 Group Routines

The following routines use the groups that are set up in MOLSCAT.

A.2.1 Informational routines

These routines return information about the given group.

character*7 function global_group(dummy)
This function returns the name of the global group.

character*7 function job_group(dummy)
This function returns the name of the working group.

character*7 function VL_group(dummy)
This function returns the name of the VL group.

integer function instance_in_group(group)
This function returns the instance number of the calling process in the group named GROUP.

integer function number_in_group(group)
This function returns the number of processes in the group named GROUP which the process is in.

A.2.2 In Group Communication Routines

There are two communication routines which exploit the connectivity of the given group

subroutine synchronous_send_next(type, buffer, size, tag, group)
integer type, size, tag
TYPE buffer(1:size)
character*7 group
Synchronous_send_next sends a message of size SIZE held in the buffer BUFFER to the process with instance number $\mathrm{mod}(i_g + 1, n_g)$ in the group named GROUP. The message is of type TYPE, valid types being defined in the file types.finc. The message is given an identification number MSG_TAG. TAG is redundant in the PVM implementation. The communication is synchronous in the sense that no further computation will occur until the message is safely on its way, but not necessarily received.

subroutine synchronous_receive_prev(type, buffer, size, tag, group)
integer type, size, tag
TYPE buffer(1:size)
character*7 group
Synchronous_receive_prev receives a message of size SIZE and puts it in buffer BUFFER from the process with instance number $\mathrm{mod}(i_g - 1, n_g)$ in the group named GROUP. The message is of type TYPE, valid types being defined in the file types.finc. The message must have an identification number MSG_TAG. TAG is redundant in the PVM implementation. The communication is synchronous. No further computation will occur until the message is received.

A.2.3 Group operations

The following routines perform operations involving data on all processes of the group. Every process must call the routine before completion and subsequent computation can occur.

The routines are all of the form
subroutine group_*TYPE* OPERATION(a, n, work, group)
integer n
TYPE a(1:n), work(1:n)
character*7 group
where

- if TYPE=dp then a double precision operation is performed. Valid operations are
  - sum - for every corresponding element of A across the named group add all the elements together. On return A holds the result of the operation.

- prod - as for sum but take the product instead.
- max - as for sum but perform a max function instead.
- min - as for sum but perform a min function instead.
- use - as for sum but perform the user defined function held in user_dpop

- if TYPE=r then a single precision operation is performed. Valid operations are as for TYPE=dp., except the user defined function is held in user_rop.
- if TYPE=i the an integer operation is performed. Valid operations are as for TYPE=dp., except the user defined function is in user_iop.
- if TYPE=l (el) then a logical operation is performed. Valid operations are
  - and - for every corresponding element of A across the named group and all the elements together. On return A holds the result of the operation.
  - or - as for and but logical or instead.
  - use - as for and but use the function defined in user_lop instead.

A few words should be said about the user defined operations. The functions must be declared as
TYPE function user_TYPEop(x, y)
TYPE x, y
If the operation is not associative random results will occur for the order in which the data is taken will not be the same for each process.

Very closely related to these routines is

subroutine group_barrier(group)
This routine loosely synchronises the given group. Each process in the group will pause until every process in the group has called this routine.

A.3 Utilities

Three utilities are provided.

integer function id_to_instnum(id)
This routine returns the instance number *in the global group* of the process with identification number ID.

integer function instnum_to_id(instance, group)
This function returns the identification number of the process with instance number INSTANCE in the group named GROUP.

subroutine close_files
It has been found that on occasions PVM fails to flush all output streams. By explicitly closing all open files this subroutine circumvents this problem.

B Job control Routines and Utilities

In the following:

- integer joblist is an array containing the job list. It is held in the large X array until sent to the host.
- integer job_number is the number of jobs found so far.
- integer length_list is the length of the job list so far.

B.1 Job List Creation and Manipulation

The following routines create the job list and turn it into the correct form:

subroutine job_control_init(lots - see code)
The driver routine for creating and manipulating the joblist.

subroutine list_all_jobs(lots - see code)
This subroutine creates a list of all the JTOT/M/Energy combinations, with the number of basis functions and processes required for each one. These are held in job list in the same format as the (optional) job control file.

subroutine add_to_list(joblist, jtot, m, energy, n,
processes, job_number)
This routine adds an entry to the end of the job list.

subroutine read_joblist(joblist, job_number, inquire, job_list_file)
logical inquire
character*72 job_list_file
This reads in the job list from the file JOB_LIST_FILE. INQUIRE is a flag which is set if this is an inquiry run.

subroutine sort_job_list(joblist , job_number, jtot_step)
This routine sorts the job list. If JTOT_STEP>0 then the list is sorted so that the smallest jobs come first, otherwise the largest are placed at the beginning of the list.

subroutine prepare_list(joblist, job_number, length_list)
This routine prepares the possibly sorted job list from list_all_jobs into a form suitable to be sent to the host.

B.2 Requesting the Next Job

The following two routines request job descriptions:

subroutine go
Start the calculation part of the computation.

subroutine request_new_job
Request the next job from the host.

B.3 Receiving and Interpreting the Job Description

The following two routines set up the job ready for propagation:

subroutine create_group(lots - see code)
Receives the new job description, interprets it and sets up the working and VL groups.

integer function my_next_energy(lots - see code)
Returns the index of the next energy to be considered by this process.

B.4 Utilities

There is only one utility:

subroutine shell_sort(n, arr, first_dim, sort_dim, direction)
integer n, first_dim, sort_dim, direction, arr(1:first_dim, 1:n)
Performs a sort using Shell's algorithm. ARR is sorted by index SORT_DIM. If DIRECTION is 1 the array is sorted into ascending order, -1 descending.

C Routines Concerned with the Distribution of VL

There are four such routines, three functions and one subroutine. In the following

- integer n is the number of basis functions
- integer number_of_i is the number of (2D) matrices held by this process
- integer matrices_this_node(1:number_of_i) is a list of which matrices are held by this process.

subroutine VLlist(n, number_of_i, matrices_this_node,
held_by_this_node)
logical held_by_this_node(1:n)
This routine calculates and returns the number and list of the (2D) matrices which are part of the VL hypermatrix and are held by this process. It must always be called. It also works out the size and form of the communications required in the propagating stage, but this is deliberately hidden from the user. Elements of HELD_BY_THIS_NODE are true if the corresponding part of VL, indexed by i, are held by this process.

integer function VLnods(n, n_potential, next_available,
upper_bound)
integer n_potential

integer next_available, upper_bound
This routine must return the minimum number of processes which VL must be distributed. N_POTENTIAL is the number of terms in the potential, NEXT_AVAILABLE is the index of the first unused element in the dynamic memory (X) array, and UPPER_BOUND is the index of the last element in the X array. At present the last two arguments are redundant. Rather the parameter MAXMEM_VL held in the include file max_VLmem.finc is used to set how much memory VL may occupy on each process.

integer function VLnumi(n)
This returns NUMBER_OF_I.

integer function VLmem(number_of_i, matrices_this_node)
This function returns the memory required to store VL on this process before weighting by the number of terms in the potential expansion. That is the memory required to store VL is given by VLmem(n)×npotl.

D The Include Files

This section outlines the contents of the include files used in the package.

D.1 SIZES.FINC

This include file contains the size in bytes of the various data types used in the package. The file as provided is for a 32 bit machine. If run on a machine with different data type sizes this file must be altered before compilation.

D.2 COMM.FINC

This is the main include file. It contains a number of parameters and common blocks that control the communications in the calculation. The file SIZES.FINC is included in this file.

integer parameter maxnod
MAXNOD is the maximum number of working processes in the calculation.

integer parameter hostmsg, job_request, next_job, message_W_base, message_p_base, global_msg, job_msg, VL_msg
These parameters define message identification numbers for various kinds of message.

logical parameter log_msg
If set all messages used in the job are recorded.

integer parameter msgfile
This is the channel number on which the message log is written.

common /msg_hst/, host_id
integer host_id
The identification number of the host.

common /msg_nod/ me, ionode
integer me
logical ionode
ME is the identification number of the process on which this job is being executed. If IONODE is set input/output is performed, otherwise it is not.

common /msg_glb/ numnod, tid, global_instnum, global_next, global_prev
integer numnod, tid(1:maxnod), global_instnum, global_next, global_prev
This common block describes the global group.

- numnod - The number of nodes in the global group
- tid - A list of the identification numbers of the processes in the global group
- global_instnum - The instance number of this process in the global group
- global_next - The identification number of the next process in the global group ring
- global_prev - The identification number of the previous process in the global group ring.

common /msg_job/ ngroup, mychum, instnum, next, prev
integer ngroup, mychum(1:maxnod), instnum, next, prev
This common block is the same as the previous one except it refers to the working group rather than the global group.

common /msg_VL/ ngroup_VL, mychum_VL, instnum_VL, next_VL, prev_VL
integer ngroup_VL, mychum_VL(1:maxnod), instnum_VL, next_VL, prev_VL
This common block is identical to msg_glb except that it refers to the VL group rather then the global group.

common /msg_str/ glbnam, grpnam, grpnam_VL
character*7 glbnam, grpnam, grpnam_VL
This common block contains the names of the various groups. Due to the ludicrous FORTRAN 77 restriction that character variables and other types may not be mixed in common, there must be a separate common block for them.

common /msg_wpr/ nwsends, nwrecvs, wsends, wrecvs, wend
integer nwsends, nwrecvs, wsends(1:4, 1:5),
wrecvs(1:4*(maxnod-1), 1:6), wend
This common block controls the communications in the routine WAVMAT. The various variables are all set up in VLLIST. They are

- nwsends - The number of messages this process sends out in one call to WAVMAT.
- nwrecvs - The number of messages this process receives in one call to WAVMAT.
- wsends - This array contains all the information relevant to sending out W to the other processes.
- wrecvs - This array contains all the information relevant for receiving the W array from other processes.
- wend - This is the index of the last used element of the W array stored in packed format.

The other two common blocks in COMM.FINC are redundant in the PVM version.

D.3 MAX_VLMEM.FINC

This include file sets up the maximum allowed memory for the VL array on any given process. SIZES.FINC is included. The only parameter of interest in this file is

integer parameter VL_megabytes
This sets the maximum storage that VL may occupy in megabytes.

D.4 TYPES.FINC

This include file sets up the reference numbers for the various data types in the communication routines. For the present implementation they must agree with those found in the PVM include file 'FPVM3.H.'