PMP Molscat

George C. McBane
Department of Chemistry
Grand Valley State University

April 3, 2005

1 Introduction

1.1 Purpose and scope

This short manual describes a "poor man's parallel" version of the Molscat inelastic scattering program. The parallel program is a modification of the serial Molscat program, version 14. It will not be useful to people unfamiliar with the serial version and its manual. It uses the MPI message passing library for parallelization. In addition, the PMP Molscat package provides a utility called `smerge` that can be used to do simple parallel calculations without any message passing library at all.

Molscat[1] was written by Sheldon Green and Jeremy Hutson. It is widely used for solving the coupled channel equations of time-independent quantum scattering theory for inelastic processes. The usual serial version of Molscat is available at `http://www.giss.nasa.gov/tools/molscat/`.

For computation of scattering cross sections, a partial wave expansion is used; solutions of the coupled channel equations are determined for many values of the total angular momentum (`JTOT` in Molscat parlance), and then the resulting S-matrix elements are combined to compute the cross sections. Furthermore, the calculation at a single `JTOT` can be broken into two or more sections corresponding to different values of a "parity" parameter `M`. (The physical meanings of `JTOT` and `M` vary with the type of calculation.) Each calculation with specified values of `JTOT` and `M` is essentially independent of all the others. The poor man's parallel ("PMP") version of Molscat parallelizes the loop over `JTOT` and `M`, distributing the required (`JTOT`, `M`) pairs across the available processes.

In the serial version of Molscat, the program completes all the propagations required, then computes the desired cross sections and prints the results in a single run. Since in the parallel version the required S-matrices are determined

by separate processes, they are not all available at once. In the PMP version, each separate process simply writes its S-matrices to a file using the ISAVEU option of Molscat. The user then runs a separate (serial) program, called `smerge`, to combine the collection of S-matrix files into a single file with the standard Molscat format. The "postprocessor" programs supplied with Molscat can then read that S-matrix file and compute cross sections as required. Postprocessors are available to compute integral (`sig_save.f`), differential (`dcs_save.f`), pressure broadening (`prbr_save.f`), or more generalized (`sbe.f`) cross sections.

1.2 Limitations

Since the PMP version only parallelizes the `JTOT` and `M` loops, it is pointless for Molscat calculations that concern only a single `JTOT`/`M` pair. Such calculations include searching for scattering resonances and convergence checking.

The loop over energies is not parallelized in the PMP version. Each process does calculations for all the total energies for each of its `JTOT`/`M` pairs. The ISCRU file that can be used to save energy-independent matrices is used in the PMP version; each process keeps its own private copy of the ISCRU file.

IOS calculations in Molscat are completed before PMP Molscat begins distributing tasks. Therefore, while IOS calculations should work, every processor will do exactly the same calculation; for IOS calculations, just stick with serial Molscat. IOS calculations are usually cheap enough not to need parallelization anyway.

Most of the normal Molscat input flags are available. The following are not, and are trapped at initialization:

- Nonzero `KSAVE`, which is not appropriate for partial wave sums.
- Automatic termination-on-convergence in `JTOT` (`JTOTU` < `JTOTL`). This very convenient feature of the serial program does not parallelize well. However, it is relatively easy to check convergence, since you can do a new run that includes only one or two more values of `JTOT`, and compare the results with and without the S-matrices from that run included in the cross section calculation.
- The LASTIN option does not work correctly in the PMP context, because ISAVEU files generated in the first pass will be overwritten in later passes.
- The IRSTRT option is not available. If a run needs to be restarted, one should simply use a new run with different `JTOTL`, `JTOTU`, `MSET`, `MHI`, and/or `ENERGY` values to generate additional S-matrices and then use the `smerge`

program to combine the old and new ISAVEU files into a single one. `Smerge` will recognize duplicate S-matrices in separate ISAVEU files and save only one into its output file, so it is usually easy to "fill in" new S-matrices in the way IRSTRT normally would.

1.3 Other Parallel Versions

A different parallel version of Molscat was written in 1993 by Ian Bush; its manual is available on the Molscat website. That version is considerably more sophisticated than this one. It parallelizes the energy loop as well as the `JTOT/M` loops by using interprocess communication to pass data between processes doing different energies for the same `JTOT/M` pair. In addition, it provides the ability to distribute the large VL array across the memory of several processes, and it collects the S-matrices and computes integral cross sections at the end of the run like the serial version.

Bush's version uses the PVM message passing library. Unfortunately, many modern computing centers do not support PVM. It would be useful to have an MPI version of his code, but none is yet available. In order to make use of current-day clusters based on MPI, I have prepared the PMP version. Several of the techniques (and a tiny amount of the code) in PMP Molscat were copied from Bush's program. The name "poor man's parallel" is a reference to the limited capability of this version in comparison to Bush's parallelization.

2 Load balancing

2.1 Available mechanisms

PMP Molscat provides three mechanisms for distributing the work among different processes.

Static assignments In the "static dispatch" version, each process generates a list of all the `JTOT/M` tasks to be done at the beginning of the job. Each process then selects a subset of the tasks to do; each process chooses a different subset, using an approach that tries to give about the same amount of work to each process. Then all the processes work on their assigned tasks until they have finished. No interprocess communication is used at all.

Dynamic assignments In the "dynamic dispatch" version, one process acts as dispatcher. It constructs a list of all the `JTOT/M` tasks to be done, then waits for the computational processes to call in asking for work. Starting with the longest tasks, the dispatcher hands out `JTOT/M` tasks to computational

processes until all of them have been done. The next time each computational process asks for work, the dispatcher sends it a "completed" message, and the computational process then does its end-of-run cleanup and exits.

True PMP The `smerge` program makes it possible to use parallel processing without any message passing harness at all. See section 7 below for details.

2.2 Controlling which mechanism is used

The main `v14pmp.f` program is the same in both the static and dynamic dispatch versions. The user chooses which version to use at link time: `pmpstat.f` contains code for the static version, and `pmpdyn.f` generates the dynamic version.

2.3 Deciding on a mechanism

On most clusters, the dynamic mechanism will work well. The static version gives similar performance on CS calculations, and usually considerably worse performance on CC calculations. However, if your cluster structure forces you to use exactly the same number of MPI processes as you have physical CPUs, the static version may be a better choice because it does not have to dedicate one CPU to act as dispatcher. See section 5 below for more details.

3 Programs

The following programs make up PMP Molscat.

`v14pmp.f` This is the main Molscat program, modified for parallelization. All the modified sections may be found easily by searching for "pmp" in the code. Other modifications from the stock serial version include

1. updated values of the physical constants,
2. removal of the `POTENL` and related routines from the main program,
3. elimination of duplicate checking in the `IVCHK` subroutine if `MXLAM` is large. (The duplicate check in its current form is very expensive for large `MXLAM`.)

`pmpstat.f`/`pmpdyn.f` These files contain routines that need to be linked with `v14pmp.f`. Only one of the two should be linked into each executable file, `pmpstat.f` for the static-dispatch version and `pmpdyn.f` for the dynamic

version. They both include the MPI interface code, a subroutine that selects the JTOT/M pairs for each process, and some other simple routines.

`smerge.f` This program reads the separate S-matrix files produced by each of the parallel processes, and combines them into a single S-matrix file in the standard Molscat ISAVEU format. It is a standalone serial program that must be run after the parallel run has completed.

The user will also need to provide a POTENL routine and associated subroutines to evaluate the potential, and link them with `v14pmp.f`. The standard ones from Molscat v. 14 may be used. In addition, at least one of the postprocessor programs mentioned above will be needed; they are available from the Molscat website.

Finally, the LAPACK and BLAS linear algebra libraries will be necessary, as with serial Molscat. While it is possible to use the stock Fortran versions of those programs available at the Molscat website or at `http://netlib.org/`, if you have big enough problems to be interested in the parallel version of Molscat you really need optimized linear algebra libraries. A good version of the BLAS is worth six processors. Some sources for optimized BLAS are listed at the BLAS FAQ, `http://www.netlib.org/blas/faq.html`. Others include ACML for AMD processors, the Altivec libraries provided by Apple for their G5, and the BLAS by Kazushige Goto for several different processors, `http://www.cs.utexas.edu/users/flame/goto/`.

PMP Molscat is not "multithreaded"; that is, it does not try to distribute the linear algebra work of a single propagation among several processors. My tests indicated the scaling with that approach was much worse than simply letting each processor work on its own JTOT/M propagations. So no OpenMP or similar shared-memory threading is used, and ordinary single-threaded BLAS libraries are appropriate.

4 Using PMP Molscat

1. Get your problem running with the serial Molscat, version 14, available at the Molscat website, `http://www.giss.nasa.gov/tools/molscat/`. If the problem is large (presumably it is, or you wouldn't be reading a parallel Molscat manual) you can get it running on the serial version for just a few low values of JTOT.

   Make sure the input file you use for the serial version sets ISAVEU > 0, does not set KSAVE > 0, has JTOTU > JTOTL, and sets LASTIN = 1 or does

not set it at all. If you are doing multiple energies, you probably want to set ISCRU > 0 as well.

2. Compile the pmp version. You need to compile and link v14pmp.f, either pmpstat.f or pmpdyn.f, your version of POTENL and its subroutines (VRTP, VSTAR, and so on; you will have these already from setting up the serial version), and the LAPACK, BLAS, and MPI libraries for your site. A couple of .fi files are read by the programs at compile time, defining COMMON blocks and integer flags.

3. Compile smerge.f. It does not call any external library routines other than standard Fortran routines, but the .fi files distributed with it need to be in the same directory as smerge.f.

4. Compile the postprocessor program(s) you need; get them from the Molscat site, http://www.giss.nasa.gov/tools/molscat/. Certainly you should get sig_save.f and compile that. If you need something other than integral cross sections, you will need additional postprocessors.

5. Save a copy of your serial-version input file with the name molscat.in. Copy your v14pmp executable, molscat.in, and any files your POTENL routines need to the the working directory for your parallel runs.

6. Run the program. In a stock LINUX/MPI environment that will mean issuing a command something like

   ```
   mpirun -np 6 v14pmp
   ```

   for six processes. You do not need to redirect the input file; the executable looks in the working directory for molscat.in.

   When the program runs it will produce one file called molscatlog.nnnn and one file called ISAVEU.nnnn for each process, where nnnn is an integer that identifies each process (running from 0 to numproc-1). It will also produce a small amount of output to the terminal, and may produce a set of ISCRU.nnnn files as well.

7. Prepare an input file for smerge.f. It has one NAMELIST block, called &MERGE. It is adequate to set just three variables in the NAMELIST block:

   ```
   &MERGE
       autoname = .true.
   ```

```
    numfiles = 6
    outputfilename = 'fort.10'
/
```

The `autoname` variable tells the program to use the default PMP Molscat names for the ISAVEU files, and the `numfiles` variable tells it how many processes you used; the `outputfilename` is a character string specifying the name of the final assembled S-matrix file. That file must not exist already; if it does, `smerge` will write out its temporary files and exit. (It might have taken months to generate an existing S-matrix file, so overwriting it is too dangerous.)

If you are combining ISAVEU files from several different runs (either of PMP Molscat or of the serial version), or combining a set of temporary files left by a previous run of `smerge`, you can leave `autoname` unset or set it to `.false.`, and then explicitly give the names of the files you want to merge as `filenames(1)='foo'`, `filenames(2)='bar'`, and so on.

8. Run `smerge`:

   ```
   smerge < smerge.input
   ```

   You should get a few messages from the program telling you of its progress, and it should produce an output file.

9. Run the `sig_save` program. See the comments at the top of the program for definitions of its NAMELIST input. It reads an unnamed file to find the S-matrices, by default opened as unit 10; on many Unix systems, naming your `smerge` output file `fort.10` will let `sig_save` find your data.

10. (Very important) Compare the integral cross sections output by `sig_save` with those you obtained with the serial version above. They may not match exactly, because the physical constants used in the PMP version are more recent than those used in the serial program, but they should all agree to at least five significant figures.

    You have now confirmed that the parallel version is working correctly for your problem, and can modify `molscat.in` to let the program do what you need.

5 Scaling properties

5.1 Recommendations for use

CS calculations For coupled states (CS) calculations, with the static version it is best to use a number of CPUs that is a divisor of the number of `JTOT/M` pairs, and a number of MPI processes equal to the number of CPUs; each pair takes about the same amount of time, so you want all the processors to have the same number of `JTOT/M` pairs to work on. With the dynamic version you want the number of physical CPUs to be a divisor of the number of `JTOT/M` pairs, and you want the number of MPI processes to be one more than the number of CPUs. If you can't run one extra process because of your cluster software, then you are probably better off with the static version.

With CS calculations, the scaling tends to be good even up to the point where there is one CPU for each `JTOT/M` pair, so long as the rules above about the number of processors per task are followed.

A small CS calculation with a total of 14 `JTOT/M` pairs (`JTOTL` = 0, `JTOTU` = 6, `JZCSMX` = 1), taking a total of 4–5 seconds, gave the scaling results shown in Table 1. These were obtained with static allocation, though the same results can be expected with dynamic allocation if it is possible to run one more MPI process than CPU.

CPUs	total time	max process time	speedup	scaling efficiency
2	4.18	2.18	1.9	95.9%
3	4.15	1.48	2.8	93.5
4	4.29	1.27	3.4	84.4
5	4.29	0.93	4.6	92.3
6	4.34	0.95	4.6	76.1
7	4.45	0.65	6.8	97.8
8	4.36	0.66	6.6	82.6
14	4.60	0.35	13.1	93.9

Table 1: Scaling results for CS calculation, 14 tasks, with static job allocation. Times are in seconds. Speedups and scaling efficiencies are slightly overestimated because they ignore extra initialization time incurred by additional processes, but that error will decrease in longer runs.

CC calculations For close-coupled (CC) calculations, the dynamic version is usually the better choice. If you are forced to use exactly one MPI process per

CPU, and you have only a few CPUs (say, four or fewer), then the static version might give better scaling.

The scaling is not as sensitive to the number of processors for CC calculations as for CS. For CC calculations you should usually use a number of CPUs that is less than about half the number of `JTOT/M` pairs; above that value, the parallel efficiency deteriorates because of poor load balancing.

I have more limited scaling data available for CC calculations. A medium-sized, 48-task CC job (`JTOTL` = 0, `JTOTU` = 23), with a total run time of about 1000 s, gave the results listed in Table 2. As expected, running the same number of processes as CPUs reduces the dynamic speedup factor by about one.

Dispatching	CPUs	MPI proc.	speedup	efficiency
Static				
	4	4	3.4	83.9%
	5	5	4.5	90.0
	7	7	6.0	85.2
	8	8	6.2	77.1
	16	16	10.3	64.6
Dynamic				
	8	8	6.9	86.8
	8	9	7.8	93.4
	16	17	14.4	90.0

Table 2: Scaling results for CC calculation, 48 tasks.

6 Requirements on user code

The POTENL routine and its subroutines used with normal Molscat can be used unchanged with PMP Molscat, with one exception. If you are using the dynamic dispatch version, any code that is used during the actual propagations (as opposed to the initialization routines) should not use the Fortran `STOP` statement; instead, it should call the `DIE` subroutine (simply `CALL DIE`) provided in `pmpdyn.f`. That routine signals the dispatcher that it is dying, so the dispatcher does not wait forever for the dead process to call in asking for another job. The same requirement holds for other code used during the propagations.

Initialization code is free to use `STOP`. All the processes, including the one that will eventually become the dispatcher, go through the same initialization, so a `STOP` encountered there will kill all processes and there will be no problem with deadlock. Similarly, code executed after the exit from the `JTOT/M` loop is

free to use `STOP` since the dispatcher is no longer waiting for messages.

You are welcome to pilfer the DIE subroutine definition from `pmpstat.f` and use it when you link your potential code with serial Molscat, so you don't have to maintain two separate versions of your potential routines.

7 Running without MPI

If you don't have an MPI-based cluster, you can still use several independent machines (which must have the same Fortran unformatted record structure) to solve a large problem. Use the serial version of Molscat, and set up one input file that would do the whole problem if given long enough. Be sure to set `ISAVEU` > 0. Then create several versions of that file, all identical except that a different range of JTOT is done by each. Run them separately (presumably one on each of your collection of machines). Gather the resulting ISAVEU files into a single directory. Make an input file for `smerge` that lists those ISAVEU files explicitly as described above, and run `smerge`. You now have a single ISAVEU file that can be read by the postprocessor programs to calculate cross sections.

8 Acknowledging PMP Molscat

If you use PMP Molscat in published work, in addition to the usual Molscat citations, please also cite "George C. McBane, "PMP Molscat", a parallel version of Molscat version 14 available at `http://faculty.gvsu.edu/mcbaneg/pmpmolscat`, Grand Valley State University (2005)."

References

[1] J. M. Hutson and S. Green. MOLSCAT computer code, version 14 (1994), distributed by Collaborative Computational Project No. 6 of the Engineering and Physical Sciences Research Council (UK).