ParGAP

(Parallel GAP)
A GAP4 Package

Version 1.1.1

by

Gene Cooperman
College of Computer Science
Northeastern University
Boston, MA, U.S.A.
email: gene@ccs.neu.edu

December 2001

Contents

1 Writing Parallel Programs in GAP Easily

The ParGAP (Parallel GAP) package provides a way of writing parallel programs using the GAP language. Former names of the package were ParGAP/MPI and GAP/MPI; the word *MPI* refers to *Message Passing Interface*, a well-known standard for parallelism. ParGAP is based on the MPI standard, and this distribution includes a subset implementation of MPI, to provide a portable layer with a high level interface to BSD sockets. Since knowledge of MPI is not required for use of this software, we now refer to the package as simply ParGAP. For more information visit the author's ParGAP home page at:

`http://www.ccs.neu.edu/home/gene/pargap.html`

For some background reading, see [Coo95] and [Coo97].

This first chapter is intended to help a new user set up ParGAP and run through some quick examples: see

- Section 1.1 for an overview of the features of ParGAP and a general discussion of how it's implemented;
- Section 1.2 for how to install ParGAP;
- Section 1.3 for how to run ParGAP (**not** by using `RequirePackage`); and
- Section 1.4 for some introductory ParGAP examples.

The later chapters present detailed explanations of the facilities of ParGAP. Because parallel programming is sufficiently different from sequential programming, this author recommends printing out at least Chapters 1 through 5, and skimming through those chapters for areas of interest, before returning to the terminal to try out some of the ideas. This document can be found in `.../pkg/pargap/doc/manual.dvi` of the software distribution. You may also want to print the index at the end of `manual.dvi`. In particular, the heading `example` in the index, or `??example` from within GAP, should be useful. If you prefer postscript, the UNIX command `dvips` will convert that file to postscript form.

The development of ParGAP was partially supported by National Science Foundation grants CCR-9509783 and CCR-9732330.

1.1 Overview of ParGAP

ParGAP is currently functional only on UNIX installations. (Cygwin for Windows is also an option, if you would like to port it.) ParGAP can be installed on top of an existing GAP installation. See Section 1.2 for instructions on installation of ParGAP. At the time that ParGAP is invoked, a special "procgroup" file must be available to tell ParGAP which processors to use for slave processors. See sections 1.2 and 1.4 for instructions on invoking ParGAP. If there are questions or bugs concerning ParGAP, please write to: `gene@ccs.neu.edu`

If one wishes only to try out the parallel features, the first five pages of this manual (through the section on the slave listener) will suffice for installation, and using it. For the more advanced user who wishes to design new parallel algorithms or port old sequential code to a parallel environment, it is strongly recommended to also read the sections following on from Section 3.

ParGAP should be invoked via the script `bin/pargap.sh` created by the installation process which invokes *GAP_ROOT_DIR*`/bin/`*ARCH*`/pargapmpi`, where *ARCH* depends on your system but is the same directory

in which the gap binary is found. MPI and the higher layers will not be available if the binary is invoked in the standard way as gap. This is a feature, since a single binary and source distribution serves both for the standard GAP and for ParGAP.

ParGAP is implemented in three layers: 1) MPI, 2) Slave Listener, and 3) Master Slave (TOP-C abstraction). Most users will find that the two highest layers (Slave Listener and Master Slave) meet all their needs.

1) MPI:

The lowest layer is MPI. Most users can ignore this layer. MPI is a standard for message-based parallel computation. A subset of the original MPI commands is provided. The syntax is modified from the original C binding to make a GAP binding in an interpreted environment more convenient. This includes default arguments, useful return values, and Error break in the presence of errors. MPI_Init() (see 7.2.2) and MPI_Finalize() (see 7.2.2) are invoked automatically by ParGAP.

The MPI layer is not documented, since most users will not be using it. From GAP level, you can type: MPI_*tabtab* to see all implemented MPI functions and variables. However, typing the symbol name alone (e.g.: MPI_Send;) will cause it to display the calling syntax. The same information is displayed after an incorrect call. The return value is typically obvious. MPI is implemented in src/pargap.c. The standard distribution uses a simple, subset implementation of MPI in pkg/gapmpi/mpinu/, which is implemented on top of a standard sockets interface. It is possible to substitute other implementations of MPI.

For those who wish to directly use the MPI interface, the meanings of the MPI calls are best found from the standard MPI documentation:

MPI Forum:

http://www.mpi-forum.org/

MPI Standard (version 1.1):

http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html

UNIX style man pages:

http://www-c.mcs.anl.gov/mpi/www/

2) Slave Listener:

This layer provides basic message passing facilities for communication among multiple ParGAP processes in a form that is more convenient for programming than the lower MPI layer. This will be the most useful entry point to ParGAP for most users. This is the default mode for ParGAP. Each remote (slave) process is in a receive-eval-send loop, in which the slave receives a GAP command from the local or master, the slave evaluates the GAP command, and the slave then sends the result back to the master as a GAP object.

Almost all commands in the slave listener are of the form *Msg* e.g. SendMsg() (see 2.1.1), RecvMsg() (see 2.1.2), ProbeMsg() (see 2.1.14). Since the slave is in a receive-eval-send loop, every SendMsg(*cmd*) on the master must be balanced by a later RecvMsg(). SendRecvMsg() (see 2.1.5) is provided to combine these steps. A few parallel utilities are also included, such as ParRead() (2.1.12), ParList() (2.1.13), ParEval() (2.1.10), etc.

Messages are arbitrary GAP objects. Note that arguments to any GAP function are evaluated before being passed to the function. Hence, any argument to SendMsg() or ParEval() would be evaluated

locally before being sent across the network. For this reason, arguments can also be given as strings, to delay evaluation until reaching the destination process. Hence, real strings must be quoted: `ParEval("x:="abc";");` Additionally, multiple commands are valid, and the final ";" of the string is optional. So, one can write:

```
BroadcastMsg("x:=\"abc\"; Print(Length(x), \"\\n\")");;
```

A full description is contained in Chapter 2.

3) Master Slave:
The Master Slave facility is provided both for writing complex parallel software, and as an easier way to parallelize previous or "legacy" sequential code. While the Slave Listener may be sufficient for simple parallel requirements, more complex software requires a higher level abstraction. The fundamental abstractions of the master slave layer are the **task** and the **shared data**.

1) The task typically corresponds to the procedure or inner body of a loop in a sequential program. This is the part that must be repetitively computed in parallel.
2) The shared data typically corresponds to the data of a sequential program that is not within the local scope of the task. Often this is a global data structure. In the case that the task is the inner body of a loop, the shared data may be a local data structure that is outside the local scope of the loop.

It is usually quite easy to identify the task and the shared data of a sequential program or algorithm, which is the first step in parallelizing an algorithm.

The Master Slave parallel model described here has also been successfully used in C and in LISP. It has been used both in distributed memory and shared memory environments, although this version in GAP currently works only in a distributed environment. In the C language, this parallel model is known as TOP-C (Task Oriented Parallel C). For examples of the use of the TOP-C model see [Coo98], [CFTY94], [CH97], [CHLM97], [CLMW96], and [CT96].

While no parallel software can eliminate the problem of designing an algorithm that is efficient in a parallel environment, the TOP-C abstraction eases the job by eliminating programmer concerns about lower level details, such as message passing, migration and replication of data, load balancing, etc. This leaves the programmer to concentrate on the primary goal: maximizing the concurrency or parallelism.

1.2 Installing ParGAP

Installing ParGAP should be relatively simple. However, since there are many interactions both with the GAP kernel and with the UNIX operating system, in a minority of cases, manual intervention will be necessary. If you are part of this minority, please see the section 1.7. The most common problem is the local security policy; ParGAP is more pleasant to use when you don't have to manually provide the password for each slave. See section 1.9 for suggestions in this respect.

To install the ParGAP package, move the file `pargap-`*XXX*`.zoo` or `pargap-`*XXX*`.tar.gz` (for some version number *XXX* of ParGAP) into the `pkg` directory in which you plan to install ParGAP. Usually, this will be the directory `pkg` in the hierarchy of your version of GAP 4 (in fact, currently it is not possible to have the `pkg` directory separate from GAP's `pkg` directory; we hope to remedy this in future versions of ParGAP so that it will also possible to keep an additional `pkg` directory in your private directories; section "ref:installing gap packages" of the GAP 4 reference manual gives details on how to do this, when it's possible.)

Now change into the `pkg` directory in which you plan to install ParGAP. If you got a `.zoo` file, unpack it with:

```
unzoo -x pargap-XXX
```

If you got a `.tar.gz` file and your `tar` command supports the `z` option, unpack it with:

```
tar zxf pargap-XXX.tar.gz
```

or otherwise unpack in two steps with:

```
gunzip pargap-XXX.tar
tar xvf pargap-XXX.tar
```

Whether you got the `.zoo` or `.tar.gz` archive you should now have a new directory `pargap`. As for a generic GAP package, do:

```
cd pargap
./configure ../..
make
```

If your version of GAP is earlier than GAP 4.3 you will first need to adjust GAP's `lib/init.g` file; see item 0. of Section 1.7.

Your ParGAP should now be ready to use. Now read the next section which decribes how to run ParGAP (if you are reading this from GAP's on-line help, type: `?>`).

1.3 Running ParGAP

After doing the `configure` and `make` steps of ParGAP's installation process (see Section 1.2), you should find in ParGAP's `bin` subdirectory a script

```
pargap.sh
```

which you should use to start ParGAP. (ParGAP can **not** be started by starting GAP 4 in the usual way, and using `RequirePackage`; doing so will result in `Info`-ed advice to read this section.) Edit the `pargap.sh` script if necessary, copy it to a standard path and rename it according to how you intend to call ParGAP (e.g. rename it: `pargap`). Also, in the `bin` subdirectory is a sample `procgroup` file which defines the master and slave processes that will be used by ParGAP. When ParGAP is started it looks for a file called `procgroup` in the current directory, unless the `-p4pg` option is used. Thus if you renamed your shell script `pargap`, the following are valid ways of starting ParGAP:

```
pargap
```

(if current directory contains the file: `procgroup`), or

```
pargap -p4pg myprocgroupfile
```

(where *myprocgroupfile* is the complete path of your procgroup file – there is no restriction on how you name it).

If you had trouble installing ParGAP, see the section 1.7. Otherwise continue onto Section 1.4 and try out ParGAP.

Note: The script `pargap.sh` defines the program that runs ParGAP as `pargapmpi`. In fact, after installation `pargapmpi` is a symbolic link to the GAP binary named `gap`. The same binary runs both GAP and ParGAP; when the binary is invoked as `gap` GAP runs in the usual way without any parallel features; only when the binary is invoked as `pargapmpi` are the parallel features incorporated. See Section 1.10 for more details.

Now you are ready to test your installation, try the example in the following section (if you are reading this from GAP's on-line help, type: `?>`).

1.4 Extended Example

After installation, try it out. Invoke ParGAP as described in Section 1.3 and try the example below (but substitute your own program where you see `"/home/gene/myprogram.g"`). The commands in this first example are also found in the `README` file. So, you may wish to copy text from the `README` file and paste it into a `ParGAP` session. If you are using the unmodified `procgroup` file, your **remote slaves** will be other processes on your local machine. It is a good idea to run only on your local machine for your first experiments and while you are debugging parallel programs. When you wish to experiment with using remote machines, you can then proceed to the following section, 1.6.

```
gap> # This assumes your procgroup file includes two slave processes.
gap> PingSlave(1); #a 'true' response indicates Slave 1 is alive
true
gap> # Print() on slave appears on standard output
gap> # i.e. after the master's prompt.
gap> SendMsg( "Print(3+4)" );
gap> 7
gap> # A <return> was input above to get a fresh prompt.
gap> #
gap> # To get special characters (including newline: '\n')
gap> # into a string, escape them with a '\'.
gap> SendMsg( "Print(3+4,\"\\n\")" );
gap> 7

gap> # Again, a <return> was input above after the 7 and new-line
gap> # were printed to get a fresh prompt.
gap> #
gap> # Each SendMsg() is normally balanced by a RecvMsg().
gap> SendMsg( "3+4", 2);
gap> RecvMsg( 2 );
7
gap> # The following is equivalent to the two previous commands.
gap> SendRecvMsg( "3+4", 2);
7
gap> # Flush any messages that are pending. The response is
gap> # the number of messages flushed. (Above, the two
gap> # SendMsg("Print...") (to the default slave: 1) did not
gap> # have a corresponding RecvMsg() command.)
gap> FlushAllMsgs();
2
gap> # As with Print() the result of Exec() appears on standard
gap> # output. Print() and Exec() are each 'no-value' functions,
gap> # and so the result of a RecvMsg() in these cases
gap> # is "<no_return_val>".
gap> SendRecvMsg( "Exec(\"pwd\")" ); # Your pwd will differ :-)
/home/gene
"<no_return_val>"
gap> # Put default slave into an infinite loop.
gap> SendMsg("while true do od");
gap> # Default slave can't execute the next command until it's
gap> # finished with the previous command.
gap> SendMsg("Print(\"WAKE UP\\n\")");
```

```
gap> # Check to see if a message is waiting to be collected but
gap> # return immediately (i.e. don't get blocked by waiting for
gap> # a message to appear). A 'false' response indicates the
gap> # infinite loop hasn't terminated and produced a value yet!
gap> ProbeMsgNonBlocking();
false
gap> # Send an interrupt to each slave, slave 1 will see the
gap> # following command and print 'WAKE UP', and then all
gap> # pending messages are flushed.
gap> ParReset();
... resetting ...
WAKE UP
0
gap> # The return value, 0, from ParReset() indicates there
gap> # were 0 pending messages flushed, confirming correctness
gap> # of ProbeMsgNonBlocking() when it returned "false"
gap> SendRecvMsg( "a:=45; 3+4", 1 );
7
gap> # Note "a" is defined on slave 1, not slave 2.
gap> SendMsg( "a", 2 ); # Slave prints error, output on master
gap>  Variable: 'a' must have a value
gap> # <return> entered to get fresh prompt.
gap> RecvMsg( 2 ); # No value for last SendMsg() command
"<no_return_val>"
gap> RecvMsg( 1 );
45
gap> myfnc := function() return 42; end;;
gap> # Use PrintToString() to define myfnc on all slave processes
gap> BroadcastMsg( PrintToString( "myfnc := ", myfnc ) );
gap> SendRecvMsg( "myfnc()", 1 );
42
gap> FlushAllMsgs(); # There are no messages pending.
0
gap> # Execute analogue of GAP's List() in parallel on slaves.
gap> squares := ParList( [1..100], x->x^2 );
[ 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256,
  289, 324, 361, 400, 441, 484, 529, 576, 625, 676, 729, 784, 841,
  900, 961, 1024, 1089, 1156, 1225, 1296, 1369, 1444, 1521, 1600,
  1681, 1764, 1849, 1936, 2025, 2116, 2209, 2304, 2401, 2500, 2601,
  2704, 2809, 2916, 3025, 3136, 3249, 3364, 3481, 3600, 3721, 3844,
  3969, 4096, 4225, 4356, 4489, 4624, 4761, 4900, 5041, 5184, 5329,
  5476, 5625, 5776, 5929, 6084, 6241, 6400, 6561, 6724, 6889, 7056,
  7225, 7396, 7569, 7744, 7921, 8100, 8281, 8464, 8649, 8836, 9025,
  9216, 9409, 9604, 9801, 10000 ]
gap> # Ensure problem shared data is read into master and slaves.
gap> # Try one of your GAP program files instead.
gap> ParRead( "/home/gene/myprogram.g");
```

Now that you have done a fairly rudimentary test of ParGAP you should be ready to do something a little bit more interesting:

```
gap> ParInstallTOPCGlobalFunction( "MyParList",
> function( list, fnc )
>   local result, iter;
>   result := [];
>   iter := Iterator(list);
>   MasterSlave( function() if IsDoneIterator(iter) then return NOTASK;
>                           else return NextIterator(iter); fi; end,
>                fnc,
>                function(input,output) result[input] := output;
>                                       return NO_ACTION; end,
>                Error
>              );
>   return result;
> end );
gap> MyParList( [1..25], x->x^3 );
master -> 1:  1
master -> 2:  2
2 -> master: 8
1 -> master: 1
master -> 1:  3
master -> 2:  4
2 -> master: 64
1 -> master: 27
master -> 1:  5
master -> 2:  6
2 -> master: 216
1 -> master: 125
master -> 1:  7
master -> 2:  8
2 -> master: 512
1 -> master: 343
master -> 1:  9
master -> 2:  10
2 -> master: 1000
1 -> master: 729
master -> 1:  11
master -> 2:  12
2 -> master: 1728
1 -> master: 1331
master -> 1:  13
master -> 2:  14
2 -> master: 2744
1 -> master: 2197
master -> 1:  15
master -> 2:  16
2 -> master: 4096
1 -> master: 3375
master -> 1:  17
master -> 2:  18
2 -> master: 5832
1 -> master: 4913
master -> 1:  19
```

```
master -> 2:  20
2 -> master: 8000
1 -> master: 6859
master -> 1:  21
master -> 2:  22
2 -> master: 10648
1 -> master: 9261
master -> 1:  23
master -> 2:  24
2 -> master: 13824
1 -> master: 12167
master -> 1:  25
1 -> master: 15625
[ 1, 8, 27, 64, 125, 216, 343, 512, 729, 1000, 1331, 1728, 2197, 2744, 3375,
  4096, 4913, 5832, 6859, 8000, 9261, 10648, 12167, 13824, 15625 ]
gap> ParInstallTOPCGlobalFunction( "MyParListWithAglom",
> function( list, fnc, aglomCount )
>   local result, iter;
>   result := [];
>   iter := Iterator(list);
>   MasterSlave( function() if IsDoneIterator(iter) then return NOTASK;
>                           else return NextIterator(iter); fi; end,
>                fnc,
>                function(input,output)
>                  local i;
>                  for i in [1..Length(input)] do
>                    result[input[i]] := output[i];
>                  od;
>                  return NO_ACTION;
>                end,
>                Error,  # Never called, can specify anything
>                aglomCount
>              );
>   return result;
> end );
gap> MyParListWithAglom( [1..25], x->x^3, 4 );
master -> 1: (AGGLOM_TASK): [ 1, 2, 3, 4 ]
master -> 2: (AGGLOM_TASK): [ 5, 6, 7, 8 ]
1 -> master: [ 1, 8, 27, 64 ]
2 -> master: [ 125, 216, 343, 512 ]
master -> 1: (AGGLOM_TASK): [ 9, 10, 11, 12 ]
master -> 2: (AGGLOM_TASK): [ 13, 14, 15, 16 ]
1 -> master: [ 729, 1000, 1331, 1728 ]
2 -> master: [ 2197, 2744, 3375, 4096 ]
master -> 1: (AGGLOM_TASK): [ 17, 18, 19, 20 ]
master -> 2: (AGGLOM_TASK): [ 21, 22, 23, 24 ]
1 -> master: [ 4913, 5832, 6859, 8000 ]
2 -> master: [ 9261, 10648, 12167, 13824 ]
master -> 1: (AGGLOM_TASK): [ 25 ]
1 -> master: [ 15625 ]
[ 1, 8, 27, 64, 125, 216, 343, 512, 729, 1000, 1331, 1728, 2197, 2744, 3375,
  4096, 4913, 5832, 6859, 8000, 9261, 10648, 12167, 13824, 15625 ]
```

If you wish an accelerated introduction to the models of parallel programming provided here, you might wish to read the beginning of Chapter 2 through section 2.1, and then proceed immediately to Chapter 3.

1.5 Author

The ParGAP package was designed and written by Gene Cooperman, College of Computer Science, Northeastern University, Boston, MA, U.S.A.

If you use ParGAP to solve a problem then please send a short email to `gene@ccs.neu.edu` about it, and cite the ParGAP package as follows:

```
\bibitem[Coo99]{Coo99}
      Cooperman, Gene,
      {\sl Parallel GAP/MPI (ParGAP/MPI)}, Version 1,
      College of Computer Science, Northeastern University, 1999,
      \verb+http://www.ccs.neu.edu/home/gene/pargap.html+.
```

1.6 Invoking ParGAP with Remote Slaves

ParGAP, unlike GAP, must be invoked under a separate name. After ParGAP has been installed, a script `bin/pargap.sh` will have been created which (after any changes you needed to make; see Section 1.2) you should use to invoke ParGAP. This is similar to *GAP_ROOT_DIR*`/bin/gap.sh` that is used to invoke the non-parallel GAP. Installers are encouraged to treat `pargap.sh` in analogy to `gap.sh`. For example, if your site has copied `gap.sh` to `/usr/local/bin/gap`, then you should also look for the `pargap.sh` script as `/usr/local/bin/pargap`.

In addition, when `pargap` (we'll assume that's how ParGAP is invoked at your site) is called, there must be a file, `procgroup`, in the current directory, or alternatively, if you wish to use a single procgroup file for all jobs, and that procgroup file is in `/home/joe`, then you can alias `pargap` to `pargap -p4pg /home/joe/procgroup`.

The procgroup file has a simple syntax, taken from the MPICH implementation of MPI (inherited from P4). A `#` in column 1 introduces a comment line. The first non-comment line should be `local 0`, verbatim. This line declares the master process as the local process. Other lines are of the form:

host-machine `1` *pargap-script*

e.g.

```
regulus.ccs.neu.edu 1 /usr/local/bin/pargap
```

The first field is the hostname for a remote process. The second field specifies one thread per process. (ParGAP recognizes only the value 1 for the second field.) The third field is an absolute pathname for ParGAP, as it would be called on the remote process. Note that you can repeat the same line twice if you want two remote ParGAP processes on the same processor. The default `procgroup` provided in the distribution will have lines of form:

`localhost 1` *path-of-provided-pargap.sh*

If you change *path-of-provided-pargap.sh* to just, say, `pargap`, this will work only if `pargap` is in your path on the remote machine shell (`localhost` in this case), using your default shell. On most machines, `localhost` is an alias for the local processor. This is a good default for debugging, so that you don't disturb users on other machines.

MPI will use a line

host-machine `1` *pargap-script*

to create a UNIX subprocess executing:

`rsh` *host-machine pargap-script*

Suppose *host-machine* is `regulus.ccs.neu.edu` and *pargap-script* is `/usr/local/bin/pargap` as in the above example, and we were to have trouble invoking ParGAP, then it would be a good idea to try invoking `rsh regulus.ccs.neu.edu` from a UNIX prompt and if that succeeds, to then try executing the full `rsh` command.

A typical problem is that the remote processor requires a password to login. MPI requires a login without passwords. Typically, `/etc/hosts.equiv` has not been set up to remove the password requirement for your remote host. Sometimes this can be solved by an appropriate `.rhosts` file in your home directory on the remote host. Sometimes, PAM is also used for user authentication (see `/etc/pam.conf`). `man in.rshd` also has helpful information. Consult your system staff for further analysis. In these days of hyper-security, `rsh` may be disabled at your site and you may have to use `ssh` instead; if so, there is a solution here: add the lines

```
##############################################################################
##
##  RSH . . . . . .  remote shell used by ParGAP
##
##
RSH=ssh
export RSH
```

before the `GAP` block with the `exec` line. (Of course, the `#` lines are not needed; they are comments.)

Note that the remote ParGAP process will not read from standard input, although signals such as SIGINT (`^C`) may be received by the remote process. However, the remote ParGAP process will write to standard output, which is relayed to the local process. So,

```
gap> SendMsg("Exec(\"hostname\")", 2);
```

will execute and print from the remote process.

1.7 Problems with Installation

If you still have problems, here is a list of things to check.

0. In versions of GAP earlier than GAP 4.3 some ParGAP "hooks" need to be added to GAP's `lib/init.g` file. Please add:

   ```
   PAR_GAP_SLAVE_START := fail;
   ```

   before the line:

   ```
   READ(GAP_RC_FILE);
   ```

   and add:

   ```
   if PAR_GAP_SLAVE_START <> fail then PAR_GAP_SLAVE_START(); fi;
   ```

   at the end of the file.

1. Do you have enough swap space to support multiple GAP processes? A simple way to check this is with the UNIX command, `top`. The Linux version of `top` sorts by memory usage if you type `M`.
2. `make` tries to automatically create:

`pkg/pargap/bin/pargap.sh`

and copy the parameters from *GAP_ROOT*`/bin/gap.sh`. *GAP_ROOT* was specified when you executed `./configure` *GAP_ROOT* to install ParGAP. This can be error-prone if your site has an unusual setup. If you execute *GAP_ROOT*`/bin/gap.sh`, does gap come up? If so, compare it with `pargap.sh` and check for correct settings in `.../pkg/pargap/bin/pargap.sh`?

3. Did ParGAP find your `procgroup` file? [It looks in the current directory for `procgroup`, or for:

   `...` `-p4pg` *PATH*`/procgroup`

   on the command line.]
4. Were the remote slave processes able to start up? If so, could they connect back to the master? To test connectivity problems, try manually starting a remote slave by executing a line in the script. Try a simple `rsh` *remote-hostname* to see if the issue is with security. If your site uses `ssh` instead of `rsh`, then there is a security issue. Read Section 1.9, and possibly `man sshd`.
5. If the previous step failed due to security issues, such as requesting a password, you have several options. `man rshd` tells you the security model at your site (or possibly `man ssh` if you use that). Then read Section 1.9.
6. Is the `procgroup` file in your current directory set correctly? Test it. If you are calling it on a remote host, manually type:

   `rsh` *HOSTNAME ParGAP*

   where *HOSTNAME* and *ParGAP* appear exactly as in `procgroup`, e.g.

   ```
   rsh denali.ccs.neu.edu /usr/local/gap4r3/bin/pargap.sh
   ```

   In some cases, `exec` is used to save process overhead. Also try:

   `rsh` *HOSTNAME* `exec` *ParGAP*

   If you plan to call it on localhost, try just: *ParGAP*

   Note that if not all the slave processes succeed in connecting to the master, then ParGAP writes out a file:

   `/tmp/pargapmpi--rsh.$$`

   where `$$` is replaced by the the process id of the ParGAP process.
7. Is `pargap` listed in `.../pkg/ALLPKG`? [It's needed to autostart slaves.]
8. Inside ParGAP, has MPI been successfully initialized? Try:

   ```
   gap> MPI_Initialized();
   ```

9. A remote (slave) ParGAP process starts in your home directory and tries to `cd` to a directory of the same name as your local directory. Check your assumptions about the remote machine. Try:

   ```
   gap> SendRecvMsg("Exec(pwd)"); SendRecvMsg("UNIX_Hostname()");
   gap> SendRecvMsg("UNIX_Getpid()");
   ```

10. If the connection dies at random, after some period of time: You can experiment with `SO_KEEPALIVE` and variants. (See `man setsockopt`.) This periodically sends **null messages** so the remote machine does not think that the originating machine is dead. However, if the remote machine fails to reply, the local process sends a SIGPIPE signal to notify current processes of a broken socket, even though there might have been only a temporary lapse in connectivity. `ssh` specifies `KeepAlive yes` by default, but setting `KeepAlive no` might get you through some transient lapses in connectivity due to high congestion. You may also want to experiment with: `setenv RSH "rsh -n"`
11. Read the documentation for further possible problems.

1.8 Problems with Hosts on Multiple Networks

If a host is on multiple networks, it will have multiple IP addresses and usually multiple hostnames. In this case, the master process cannot always guess correctly which IP address (which internet address) should be passed to the slave process, so that the slave process can call back to the master. In such cases, you may need to tell ParGAP which hostname or IP address to use for the callback. This is done by setting the UNIX environment variable, `CALLBACK_HOST`, as in the example below.

```
# [ in sh/bash/... ]
CALLBACK_HOST=denali.ccs.neu.edu; export CALLBACK_HOST
# [ in csh/tcsh/... ]
setenv CALLBACK_HOST=denali.ccs.neu.edu
```

The appropriate line for your shell can be placed in your shell initialization file. Alternatively, you can set this up for all users by placing the Bourne shell version (for `sh`) somewhere between the first and last line of `.../pkg/pargap/bin/pargap.sh`.

1.9 Problems with Passwords (Getting Around Security)

There is a simple test to see if you need to read this section. Pick a remote machine, *HOSTNAME*, that you wish to execute on, and type: `rsh` *HOSTNAME*. If this did not work, also try `ssh` *HOSTNAME*. If you were asked for your password, then you and your system administrator may need to talk about security policy. If you were successful with `ssh` and not with `rsh` then set the environment variable, `RSH`, to the value `ssh`, as described in item 3 below.

(1) Ask your systems administrator to put the machines in a `hosts.equiv` file, so that logging in from one to the other does not require a password. (`man hosts.equiv`)

(2) Add a `.rhosts` file to your home directory (or `.shosts` for `ssh`).

(3) Hack around the problem: By default, the startup script uses `rsh` to start remote processes. However, if the environment variable `RSH` was set, the script uses the value of the environment variable instead of `rsh`. This may be useful, if you have your own script, `myrsh`, that automatically gets around the security issues. Then just type:

```
RSH=myrsh; export RSH  # [ in sh/bash/... ]
setenv RSH myrsh       # [ in csh/tcsh/... ]
```

The appropriate line for your shell can be placed in your shell initialization file. Alternatively, you can set this up for all users by placing the Bourne shell version (for `sh`) somewhere between the first and last line of `.../pkg/pargap/bin/pargap.sh`. (The example for `ssh` was given earlier.)

(4) `ssh`: `man ssh` mentions some possibilities for giving the password the first time, and then having ssh remember that future logins to that machine are authorized for the duration of the session. Don't overlook the use of `$HOME/.ssh/config` to set special parameters, such as specifying a different login name on the remote machine. Some parameters of interest might be `KeepAlive`, `RSAAuthentication`, `UseRsh`. You may also find useful information in `man sshd`.

(5) After starting ParGAP, manually call

```
/tmp/pargapmpi--rsh.$$
```

and repeatedly type in the password for each slave process. If you find yourself doing this, you may want to talk with your system administrator, since it actually hurts system security to have you repeatedly typing passwords with a concommitant risk that someone else will find out your password.

1.10 Modifying the GAP kernel

Note that this package modifies the GAP `src` and `bin` files, and creates a new GAP kernel. This new GAP kernel can be shared by traditional users of the old, sequential GAP kernel, and by those doing parallel processing.

The GAP kernel will have identical behavior to the old GAP kernel when invoked through the `gap.sh` script or the `bin/@GAParch@/gap` binary. The new ParGAP variables will appear to the end user **ONLY** if the GAP binary was invoked as `pargapmpi`: a symbolic link to the actual GAP binary. The script, `pargap.sh`, does this.

So, in a multi-user environment, traditional users can continue to use `gap.sh` without noticing any difference. Only an invocation of `pargap.sh` will add the new features.

In a future version of GAP, it is hoped that the GAP kernel will have enough "hooks", so that no modification of the GAP kernel is required. At that time, it will also be possible to speed up the startup time for ParGAP. Much of the startup time is caused by waiting for GAP to read its library files. It will be possible to use the GAP function, `SaveWorkspace()` to save a version with the GAP library pre-loaded. That saved version can then be used to start up ParGAP. This is not currently possible, because ParGAP needs to get at the command line of GAP before the GAP kernel sees it.

Comments and contributions to a ParGAP user library, or any other type of assistance, are gratefully accepted.

Gene Cooperman `gene@ccs.neu.edu`

2 Slave Listener

ParGAP implements a model of a slave process as a **slave listener**. This means that the slave is running a simple program:

```
(1) Read message from master [as string]
(2) Evaluate message and return result
(3) Send message to master with result [as string]
(4) Goto step 1
```

An example using this interactive style is contained in section 1.4.

Some enhancements to this simple model should also be noted. The reply message from the slave will wait in a queue until the master process decides to read it. If unwanted messages accumulate in the queue, the master can execute `FlushAllMsgs()` (see 2.1.8). If a slave process prints to the standard output, this will be visible at the console of the master process. If a slave process executes an `Error` and goes into a break loop, then it will automatically return to the top level, return any error message to the master process, and wait for another message from the master process. If a slave process goes into an infinite loop, the master process can call `ParReset();` (see 2.1.16) to interrupt all slave processes and return them to their top level loop as a slave listener.

At this point, you may wish to review the commands by looking again at the extended example in section 1.4. Note also some naming conventions:

`MPI_...`:
: A command prefix of `MPI_` signifies a GAP binding of an MPI function. These functions are low level functions on which the rest of ParGAP is built. They can be safely ignored by the casual user. (Recall that MPI, **Message Passing Interface**, is a standard for message passing.) In ParGAP, type `MPI_<`*tab*`>` for a list of all such functions.

`UNIX_...`:
: Commands with prefix `UNIX_` are additional system commands that were not present in the unmodified GAP kernel. They are typically GAP versions of UNIX commands that make life easier. `UNIX_Nice()` is an example. In ParGAP, type `UNIX_<`*tab*`>` for a list of all such functions.

`Par...`:
: Commands beginning with `Par` are "parallel" commands that should only be called by the master process. Such commands invoke all slave processes to do their work. In ParGAP, type `Par<`*tab*`>` for a list of all such functions.

2.1 Slave Listener Commands

The slave listener commands are implementd in `slavelist.g` in ParGAP's `lib` directory. Most procedures are short, and can also be read online by using GAP's `Print` command, e.g. try: `Print(SendMsg, "\n");` (the newline is needed only to get back to a clean "`gap>` " prompt). The code of `SlaveListener` and `CloseSlaveListener` (try: `Print(SlaveListener, "\n");` and `Print(CloseSlaveListener, "\n");`) is also instructive and should provide some insights into the behavior of the slave listener. Examples of slave

listener commands can be found in context in the section 1.4. Some of these commands are based on MPI. Further information on basic concepts of MPI can be found in section 7.1, but that section can be safely ignored on a first reading.

1 ▶ `SendMsg(` *command*`[,` *dest*`[,` *tag*`]] )` F

sends *command* to *dest* (a non-negative integer that is the "rank" of the destination process); *command* should normally be a string (otherwise it is evaluated on the master before being passed to *dest* which almost certainly will defeat the purpose of using `SendMsg()`). If *dest* is omitted it defaults to 1 (the rank of the first slave) on the master process (i.e. if `IsMaster()` is `true`), or to 0 (the rank of the master) on a slave process (i.e. if `IsMaster()` is `false`). The argument *tag*, if given, should be a positive integer less than 1000. The default value of *tag* is 1. Tags of value 1000 and above are reserved for use by ParGAP itself, and should not be used by application routines.

2 ▶ `RecvMsg( [`*source*`] )` F

gets a response from a command. The default value of *source* is `MPI_ANY_SOURCE`, which receives the next available message from any source. `GetLastMsgSource()` (see 2.1.3) allows one to determine the source in such cases. `GetLastMsgTag()` (see 2.1.4) always allows one to determine the tag, although most applications can ignore the tag. Tags are applied to commands by `SendMsg()` or `SendRecvMsg()` (see 2.1.1).

3 ▶ `GetLastMsgSource()` F

returns the source of the last message that was either received (e.g. by `RecvMsg()`; see 2.1.2) or simply probed (e.g. by `ProbeMsg()`; see 2.1.14).

4 ▶ `GetLastMsgTag()` F

returns the tag (see 2.1.1) of the last message that was either received (e.g. by `RecvMsg()`; see 2.1.2) or simply probed (e.g. by `ProbeMsg()`; see 2.1.14).

5 ▶ `SendRecvMsg(` *command*`[,` *dest*`[,` *tag*`]] )` F

This command is equivalent to `SendMsg(` *command*`[,` *dest*`[,` *tag*`]] ); RecvMsg([`*dest*`]);` (see 2.1.1 and 2.1.2), except that even if *dest* is omitted the source for the `RecvMsg()` part of the command always matches the destination to which *command* is sent.

Note: The response obtained will not be the response of the *command* itself if there are messages waiting to be received at the destination of *command* at the time `SendRecvMsg()` is called.

Also note that tag values of 1000 and higher are reserved for use by ParGAP.

6 ▶ `BroadcastMsg(` *command* `)` F

executes *command* on (all) slaves only. The slaves do not send back a return value.

Note: this use of the term **broadcast** is distinct from the MPI usage. In MPI, a broadcast message will be received by every process, including the process sending the message.

7 ▶ `IsMaster()` F

returns `true` if at console (i.e. if `MPI_Comm_rank() = 0`), and `false` otherwise.

8 ▶ `FlushAllMsgs()` F

flushes all messages that are waiting to be received and returns the number of messages flushed. (If there are no waiting messages 0 is returned.) It is essentially equivalent to executing `RecvMsg();;` until there are no more messages waiting to be received (see 2.1.2), except that it also returns the number of messages flushed.

9 ▶ `PingSlave(` *dest* `)` F

Check if slave *dest* is alive and listening for messages, where *dest* is a positive integer.

10 ▶ `ParEval( stringCmd )` F

Evaluate *stringCmd* on all processes, where *stringCmd* is a command inside double quotes so that it is passed as a string (like `BroadcastMsg()` (see 2.1.6), but `ParEval()` (see 2.1.10) also executes on the master and also returns a value based on result on the master.)

11 ▶ `PrintToString( object [, ...] )`

[Note that `PrintToString("abc") => "abc"` (like `Print()`, NOT `""abc""`) Hence, a useful idiom is: `ParEval( PrintToString( "foo := ", foo ) );`]

12 ▶ `ParRead( filename )` F
▶ `ParReread( filename )` F

are parallel analogues of the GAP `Read` and `Reread` functions, respectively (see "ref:read" and "ref:reread" in the Reference Manual). `ParRead` (resp. `ParReread`) executes `Read` (resp. `Reread`) on all processes. Note that it is redundant (and often incorrect) to call `ParRead` on a file that itself contains `Par...` functions. One should either place sequential functions in a file and call `ParRead` or place `Par...` functions in a file and call `Read` from the master. As an example, in writing this code, the author (after having started ParGAP from its `bin` directory via `pargap.sh`) found it useful to edit `masslave.g` in ParGAP's `lib` directory and then type `ParReread("../lib/masslave.g");`.

13 ▶ `ParList( list, func )` F

is the parallel analogue of GAP's two-argument `List` function. But faster since it also uses the slave processes.

14 ▶ `ProbeMsg( [source] )` F

probes for a pending message from *source* or any source if the argument *source* is omitted. It will block until such a message appears, and then return `true`. `^C` (interrupt) works to unblock it.

Note: When the argument *source* is omitted, `ProbeMsg` sets *source* to `MPI_ANY_SOURCE` (which is `-1`), which specifies a probe for a message from any source.

15 ▶ `ProbeMsgNonBlocking( [source] )` F

Exactly like `ProbeMsg`, but non-blocking. It returns immediately with `true` or `false`, depending on whether a message was present from *source*. The default value of *source* is `MPI_ANY_SOURCE`.

16 ▶ `ParReset()` F

flushes all pending messages from slaves, resets the slaves, pings the slaves and returns the number of messages flushed.

17 ▶ `ParBindGlobal( gvar, value )` F

Not currently implemented, due to certain technical considerations.

18 ▶ `ParDeclareGlobalValue( string )` F
▶ `ParDeclareGlobalFunction( string )` F

Similar to corresponding GAP functions. Note that unlike GAP's `DeclareGlobalFunction` and `ParDeclare-GlobalValue`, these functions also allow you to re-declare an old function or variable. The net effect is to remove the old value, and allow one to again call `InstallGlobalFunction` and `InstallValue`. This eliminates the necessity for `Reread()` in ParGAP, and it also makes it easier to place the commands in a local file, and using a simple `Read()` instead of `ParRead()`. It also makes it easier to interactively re-declare and re-install functions.

19 ▶ `ParInstallValue(` *gvar*`,` *value* `)` F
▶ `ParInstallValue(` *string*`,` *value* `)` F
▶ `ParInstallGlobalFunction(` *gvar*`,` *function* `)` F
▶ `ParInstallGlobalFunction(` *string*`,` *function* `)` F

Note that the second version of `ParInstallGlobalFunction` (with *string*) is equivalent to

```
ParDeclareGlobalFunction( string );
ParInstallGlobalFunction( gvar, function );
```

where *gvar* is a GAP variable whose name is *string*.

Note that `ParInstallValue` is currently implemented only in the version for *string*, due to certain technical considerations.

This completes the middle layer of ParGAP. It allows one to easily use parallelism interactively. There are now two choices for further reading. The recommended choice for writing your own parallel applications is to read the next chapter on the TOP-C task-oriented model of parallelism, and the follow-on chapter, containing a tutorial on the TOP-C model. These two chapters should provide enough background to write significant parallel applications. If on the other hand you are interested in MPI and the low-level fundamentals of message passing for parallel applications, then you should read Chapter 7.

3 Basic Concepts for the TOP-C model (MasterSlave)

TOP-C stands for **Task-Oriented Parallel C** [Coo96]. The "TOP-C model" is the specific master slave model implemented here. That model has been adapted for use in ParGAP. The implementation is in `masslave.g` in ParGAP's `lib` directory. Note that the functions and variables with names `TOPC...` are intended as internal functions only, and should not be used by the GAP programmer.

For the impatient, you may type `MSexample();` in a ParGAP session now. If you prefer further hands-on learning in a tutorial style, you may wish to next read Chapter 5. Eventually, if you wish a deeper understanding of the TOP-C model, you will need to read this current section and those that follow.

The initial GAP process is the **master** process, and all others are **slave** processes. It allows most of the CPU-intensive computations to be carried out on slave processes, which typically reside on remote processors. A well-developed TOP-C application should find that the master process is almost never busy when a slave process is idle, waiting for a new computation to carry out. This provides a natural way of maximizing utilization and load balancing.

The TOP-C model depends on three concepts:

the **task**:
: a function that takes an arbitrary object as its single argument, reads some or all of the global shared data, and then returns an arbitrary object as its value. The task typically corresponds to the inner loop of a typical application.

the **shared data**:
: global data, shared among all processes. This data can be read as part of the computation of a task. However, after initialization of the shared data, this data must be written (modified) **only** by a particular user-provided application routine, *UpdateSharedData*`()`.

the **action**:
: After the output of a task has been produced, an application routine must choose one of four actions to determine how the output is used.

The **task input** is defined to be the argument of the **task** (considered as a function), and the **task output** is the return value of the **task**.

3.1 Basic TOP-C (Master-Slave) commands

There is only one core TOP-C command, a utility function, and several constants. A TOP-C command must be evaluated on the master and on all slaves. We shall describe the commands in detail in the following sections, but a short list of the essentials and a small example will be helpful to set the context.

1 ▶ `MasterSlave(` *SubmitTaskInput*`,` *DoTask*`[,` *CheckTaskResult*`[,` *UpdateSharedData*`[,` *taskAgglomCount*`]]] )` F

See Section 3.2 for a description of `MasterSlave`.

2 ▶ `NOTASK` V

3 ▶ `NO_ACTION` V
▶ `UPDATE_ACTION` V
▶ `REDO_ACTION` V
▶ `CONTINUATION_ACTION(` *taskContinuation* `)` F

`CONTINUATION_ACTION()` is described in Section 3.6.

4 ▶ `IsUpToDate()` F

5 ▶ `ParInstallTOPCGlobalFunction(` *string*, *function* `)` F
▶ `ParInstallTOPCGlobalFunction(` *gvar*, *function* `)` F

A short example shows one possible implementation of `ParList()`.

```
gap> ParInstallTOPCGlobalFunction( "MyParListWithAgglom",
> function( list, fnc )
>   local result, i;
>   result := []; i := 0;
>   MasterSlave( function() if i >= Length(list) then return NOTASK;
>                           else i := i+1; return i; fi; end,
>                fnc,
>                function(input,output) result[input] := output;
>                                       return NO_ACTION; end,
>                Error
>              );
>   return result;
> end );
```

(Of course rather than type such code in a ParGAP session it's generally more convenient to have it in a file and `Read` it in.)

3.2 Other TOP-C Commands

A master-slave computation is invoked when a GAP program issues the command `MasterSlave()`. As given earlier, the typical form is:

`MasterSlave(` *SubmitTaskInput*, *DoTask* `[,` *CheckTaskResult*`[,` *UpdateSharedData*`[,` *taskAgglom*`]]]` `)`

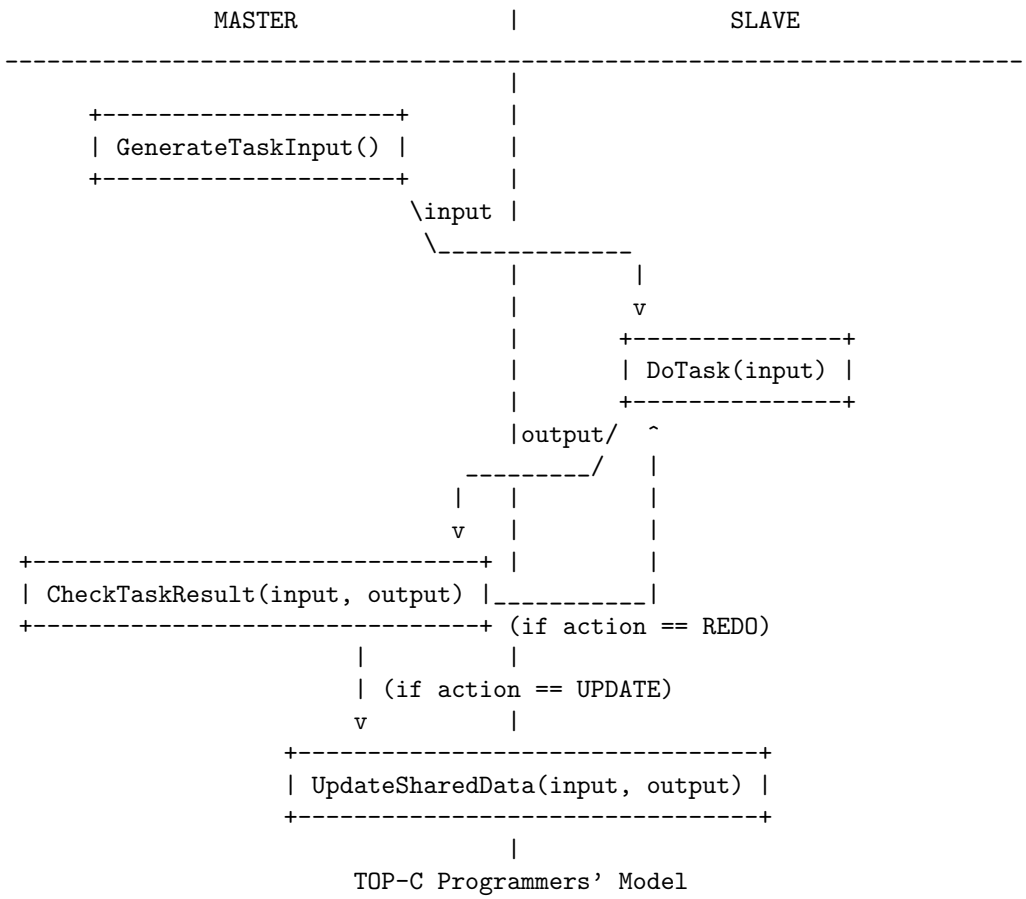
where the first four arguments of `MasterSlave()` are also functions, but they must be defined by the application writer. Their calling syntax is defined by the following GAP code, which also provides a simplified description of how a sequential (non-parallel) `MasterSlave()` would invoke these functions if there were only a single process. (A more sophisticated version of this routine is provided in ParGAP to allow one to debug within a single process first.) The use of the fifth argument, *taskAgglom*, is deferred until section 5.7.

In this section, we define `MasterSlave()` and describe the use of its four arguments in a purely sequential environment. The issues of parallelism and passing of messages between processes is covered in the next section. The call to `MasterSlave()` in ParGAP, above, will have the same result as if `MasterSlave()`

were defined equivalently to `SeqMasterSlave()` below, and then run in a standard, sequential GAP (a single process). The next section describes the multi-process implementation of `MasterSlave()` in ParGAP, in which `taskInput` is computed on the master process and sent as a message to a slave process, while `taskOutput` is computed on a slave process and sent as a message to the master process.

```
SeqMasterSlave :=
  function(SubmitTaskInput, DoTask, CheckTaskResult, UpdateSharedData)
  local taskInput, taskOutput, action;
  while true do
    taskInput := SubmitTaskInput();
    if taskInput = NOTASK then break; fi;
    repeat
      taskOutput := DoTask( taskInput );
      action := CheckTaskResult( taskOutput, taskInput );
    until action <> REDO_ACTION;
    if action = UPDATE_ACTION then
      # Modify the shared data (global data structures) here
      # Called on all processes, master and slaves
      UpdateSharedData( taskOutput, taskInput );
    fi;
  od;
end;
```

One can also follow the life of a single task in a multi-processing environment through the diagram below.



`(Life Cycle of a Task)`

Although not explicit in the code, the application writer should add comments to define the shared data. The **shared data** is defined as a global data structure that is treated as "read-write" by *UpdateSharedData*`()`, while being treated as "read-only" by *SubmitTaskInput*`()`, *DoTask*`()`, and *CheckTaskResult*`()`. Note also that an application writer may use different names for the four functions *SubmitTaskInput*`()`, etc. It is only a convention within this manual to give those functions the names, above. Similarly, *taskInput*, *taskOutput* and *action* are the conventional names used in this manual, and a given application may use different names.

In a correct ParGAP application, the shared data should be initialized to the same value on all processes before the application calls `MasterSlave()`. `MasterSlave()` is then called on all processes. After that, the shared data can be modified only by a call to *UpdateSharedData*`()`, and `MasterSlave()` arranges for each call to *UpdateSharedData*`()` to be executed on all processes. Further, *UpdateSharedData*`()` has access only to *taskInput*, *taskOutput*, and the previous value of the shared data. Thus, `MasterSlave()` maintains the same shared data uniformly on all processes.

3.3 Simple Usage of MasterSlave()

This section is concerned with formal definitions for the routines associated with ParGAP. It is important to keep in mind the pseudo-code of Chapter 3. Since `MasterSlave()` uses all the ParGAP processes, the user must invoke it on all processes. This is typically done through some function provided by the slave listener layer, such as `ParEval()` (see 2.1.10). It may be instructive for the reader to run ParGAP and type `MSexample();` now, or else to look at some examples of ParGAP applications in the section 5. This demonstrates the use of `MasterSlave()` in a typical session.

The four functions written by the application writer are: *SubmitTaskInput*`()`, *DoTask*`()`, *CheckTaskResult*`()`, and *UpdateSharedData*`()`. *DoTask*`()` is executed on a slave. *SubmitTaskInput*`()` and *CheckTaskResult*`()` are executed on the master, where a *taskInput* is generated and a corresponding *taskOutput* is received. Finally, *UpdateSharedData*`()` is executed on all processes. ParGAP arranges to automatically pass *taskInput* and *taskOutput* between the master and a slave.

Since the single master process is responsible for generating all *taskInput*s and receiving all *taskOutput*s, it is critical that computation on the master process should not become a bottleneck for a well-designed ParGAP application. Accordingly, the application writer should arrange for *SubmitTaskInput*`()` and *CheckTaskResult*`()` to execute quickly, even if this means additional computation by *DoTask*`()` or *UpdateSharedData*`()`.

As seen in the examples, *SubmitTaskInput*`()` may use global variables on the master to "remember" the last *taskInput* or other state information. Note that such global variables cannot be part of the shared data, since they are modified outside of *UpdateSharedData*`()`.

3.4 Efficient Parallelism in MasterSlave() using CheckTaskResult()

It is instructive to review the logic for the lifetime of a task, as described by the pseudo-code for `SeqMasterSlave` in Section 3.2. Initially, `MasterSlave()` calls *SubmitTaskInput*`()` on the master, which returns an application-defined GAP object, *taskInput*. `MasterSlave()` then copies *taskInput* to an arbitrary slave process, and `MasterSlave()` then calls *DoTask*`(` *taskInput* `)` on the slave. This returns an application-defined GAP object, *taskOutput*, which `MasterSlave()` copies to the master process. On the master, `MasterSlave()` then calls *CheckTaskResult*`(` *taskInput*`,` *taskOutput* `)`, which returns an action. (Recall that *taskInput*, *taskOutput* and *CheckTaskResult*`()` are defined by the application writer, and so an application program may give them different names.)

There are four possible **actions** (ParGAP constants): `NO_ACTION`, `UPDATE_ACTION`, `REDO_ACTION`, `CONTINUATION_ACTION(` *taskContinuation* `)`. A standard language idiom in ParGAP is to define *CheckTaskResult*`()` as the ParGAP function `DefaultCheckTaskResult()`, whose code is as follows:

```
DefaultCheckTaskResult := function( taskOutput, taskInput )
  if taskOutput = false then return NO_ACTION;
  elif IsUpToDate() then return UPDATE_ACTION;
  else return REDO_ACTION;
  fi;
end;
```

In the simplest case, *CheckTaskResult*() returns NO_ACTION, in which case there is no further computation related to the original *taskInput*. *CheckTaskResult*() may record global information on the master process, based on the *taskOutput*, but the shared data, and hence the state of the slave processes, will not be modified.

In the second most common case, *CheckTaskResult*() returns UPDATE_ACTION. This action causes MasterSlave() to call *UpdateSharedData*(*taskOutput*, *taskInput*) on all processes (master and slaves). This is the **only** way in which the shared data can be modified by a correct ParGAP program.

In the third most common case, *CheckTaskResult*() returns REDO_ACTION. When a REDO_ACTION action is generated, the value of *taskInput* is re-sent to the same slave that executed *DoTask*(*taskInput*) for the current task. An application will typically invoke REDO_ACTION if the shared data has changed, and this changed shared data will produce a new *taskOutput*. As before, *DoTask*() then returns a new value of *taskOutput*. Then, *taskInput* and the new *taskOutput* are again passed to *CheckTaskResult*().

Note that MasterSlave() guarantees that REDO_ACTION causes the task to be re-sent to the same slave process. This allows the application to cache in a global variable some information computed by the first invocation of *DoTask*(). A second invocation of *DoTask*() caused by the REDO_ACTION allows the task to test if the *taskInput* is the same as the last invocation. In that case, the application-defined *DoTask*() routine can recognize that this is a REDO_ACTION, and it can take advantage of the cached global variable to avoid re-computing certain quantities that would not be changed by the altered shared data. In order to make this strategy possible, MasterSlave() also guarantees that in the case of REDO_ACTION, the slave process will not have seen any intervening calls to *DoTask*() with values of taskInput other than the current value.

In typical usage, the application-defined routine, *CheckTaskResult*(), will first call IsUpToDate(). IsUpToDate() tests if the shared data has been modified since the current *taskInput* corresponding to *CheckTaskResult*() was originally generated by *SubmitTaskInput*(). The times of the relevant events are recorded as when seen on the master process. It is an error to call IsUpToDate() outside of a call to *CheckTaskResult*() by MasterSlave(). IsUpToDate() returns a boolean value, true or false.

The last possible action, CONTINUATION_ACTION(*taskContinuation*), is provided for unusual cases. As with advice about the use of "goto", it is recommended to avoid CONTINUATION_ACTION() where possible.

A favorite aphorism of this author is, "The source code is the ultimate documentation". With this in mind, the reader may also wish to read lib/masslave.g, for which readability of the code was one of the design criteria.

3.5 Modifying Task Output or Input (a dirty trick)

At this point, it should be noted that it explicitly **is** allowed to modify the input or output of a task from within *CheckTaskResult*(). This is not recommended in general, but there may be times when *CheckTaskResult*() returns an UPDATE_ACTION and must also be used to pass additional information to *UpdateSharedData*(). In order to modify a previous input or output, it is important that the application has chosen a representation of the input or output as a list or record, which can be modified in place, such that the code excerpt succeeds without error.

```
oldOutput := taskOutput;
# Modify taskOutput here
if ( IsIdenticalObj( oldOutput, taskOutput ) ) = false then
  Error( "MasterSlave() will see only oldOutput, not current taskOutput" );
fi;
return UPDATE_ACTION;
```

In principle, a **dirty trick** like this would also work in the case of returning a `REDO_ACTION`. However, this is not recommended. For that functionality, the code will be clearer if an explicit `CONTINUATION_ACTION(` *modifiedTaskOutput* `)` is returned. See Section 3.6 for further discussion on the use of `CONTINUATION_ACTION()`.

3.6 The GOTO statement of the TOP-C model

1 ▶ `CONTINUATION_ACTION(` *taskContinuation* `)`

The `CONTINUATION_ACTION()`, like the *goto* statement, is not recommended for ordinary programs, but it may be useful in unusual circumstances. This is a parametrized action. When the application routine *CheckTaskResult*`()` returns this action, `MasterSlave()` guarantees to invoke *DoTask*`()` on the same slave process as for the original task. There will have been no intervening calls to *DoTask*`()` on that slave, although there may have been an intervening call to *UpdateSharedData*`()` on that slave.

This action allows arbitrary, repeated communication between the master and a single slave process. The slave process executes *DoTask*`(` *taskInput* `)` and communicates with the master by returning a *taskOutput*. The master process executes *CheckTaskResult*`(` *taskInput*`,` *taskOutput* `)` and returns a *taskContinuation*. The original slave process then receives another call to *DoTask*`(` *taskInput* `)`, this time with *taskInput* bound to *taskContinuation*.

3.7 Being nice to other users (Nice, Alarm and LimitRss)

When you are running a long job on a network of workstations, you will often be sharing it with others. Making your parallel job as unintrusive as possible will leave you with a warmer welcome the next time that you want to use that network of workstations. Accordingly, three useful functions are provided.

1 ▶ `UNIX_Nice(` *priority* `)` F

This is similar to the `nice` command of many UNIX shells. UNIX priorities are in a range from −20 to 20 with −20 being the highest. Users typically start at priority 0. You can give yourself a lower priority by specifying a priority of 5, for example. Usually, priorities 19 and 20 are **absolute** priorities. Any process with a priority higher than 19 that wishes to run will always have precedence. Other priorities are **relative** priorities. Your process will still receive some CPU time even if other processes with higher priorities are running. You can set your priority lower, but you cannot raise it back to its original value after that. The return value is the previous priority of your process.

2 ▶ `UNIX_Alarm(` *seconds* `)` F

This causes the process to kill itself after that many seconds. This is a useful safety measure, since it is unfortunately too easy for a runaway slave process to continue if the master process is killed without the normal `quit;`. You might consider adding something like `UNIX_Alarm( 25000 );` (about 6 hours) to your `.gaprc` file. Executing `UNIX_Alarm( 0 );` cancels any previous alarm. The return value is the number of seconds remaining under the previous setting of the alarm.

3 ▶ `UNIX_LimitRss(` *size* `)` [= setrlimit(RLIMIT_RSS, ...)] F

Many dialects of UNIX (and their shells) offer a `limit` or `ulimit` command to limit the resources available to the shell. This command limits the size of the RSS (resident set size), or the amount of physical RAM used by your process. The size limit is in bytes. Unfortunately, some UNIX dialects may not allow or even silently ignore this request to limit the RSS. A UNIX command such as `top` can show you if your process RSS is staying below your requested limit.

3.8 Converting legacy sequential code to the TOP-C model

The (tutorial contains a section 5.8, about *raw* version of `MasterSlave()` that is useful for converting legacy sequential code to the TOP-C model. However, that model is not recommended for writing new code, for stylistic reasons.

4 Tutorial

Section 4.1 covers trivial parallelism (the simplest and most common form of parallel application). This is followed in Section 4.2 by a description of how to use ParGAP interactively, and Section 4.3 illustrates these principles with a short implementation of parallel streaming. Streaming refers to simultaneously running different algorithms for the same problem in different "streams" or processes, and accepting the first answer that returns. The remaining processes for the other algorithms are then terminated. ParGAP allows those processes to reside on a single CPU or on different CPUs. Hence, in the case of streaming, ParGAP is potentially useful even if one has only a single computer available, since multiple streams can still reside locally in separate processes.

For more sophisticated (non-trivial) parallel applications, the TOP-C model of parallelism is recommended, and it is described in Section 4.4.

4.1 Trivial Parallelism

In parallel computation, perhaps 80% of the applications fall under the heading of "trivial parallelism". These are situations in which one must compute many unrelated cases. For example, perhaps 200 different cases must be computed and results returned for each case and each case has no interaction with any other one. In the absence of shared data, it would be common to start 20 GAP jobs on 20 distinct workstations in a student computer laboratory. If there are 200 cases, then one writes 20 different "batch jobs", each batch job handling 10 distinct cases.

ParGAP provides strong support for this common situation. Conceptually, one is always talking to ParGAP on a master process (the local workstation), and the master process talks to each of various slave processes. In its simplest form, one merely generates a list of inputs for the cases, and writes a suitable function to provide the case information. Effectively, trivial parallelism can be expressed by a parallel version of GAP's `List()` function (see "ref:list" in the Reference Manual), which is called `ParList()` (see 2.1.13) in ParGAP.

The following is an example of a ParGAP session that uses the `ParList()` function. Observe the use of `BroadcastMsg()` (see 2.1.6) to ensure that the function `AnalyzePrimitivePermGroupsOfOrder()` is known to the slaves.

```
gap> AnalyzePrimitivePermGroupsOfOrder := function(ord)
>      #returns a list of data records of all primitive
>      #permutation groups of order ord that are represented
>      #as permutation groups in GAP's primitive group database
>
>      local PrimitiveGroupsOfOrder, AnalyzePrimitivePermGroup;
>
>      PrimitiveGroupsOfOrder :=
>        ord -> List( [1..NrPrimitiveGroups(ord)],
>                     i -> PrimitiveGroup(ord,i) );
>
>      AnalyzePrimitivePermGroup :=
>        grp -> rec( order := Size(grp),
```

```
>                    degree := NrMovedPoints(grp),
>                    baseSize := Size(BaseOfGroup(grp)) );
>
>      return List( Filtered( PrimitiveGroupsOfOrder(ord),
>                             IsPermGroup ),
>                   AnalyzePrimitivePermGroup );
>    end;;
gap> #Define AnalyzePrimitivePermGroupsOfOrder on slaves
gap> BroadcastMsg(
>      PrintToString("AnalyzePrimitivePermGroupsOfOrder := ",
>                    AnalyzePrimitivePermGroupsOfOrder ) );
gap> ParList( [2..256], AnalyzePrimitivePermGroupsOfOrder );
[ [ rec( order := 2, degree := 2, baseSize := 1 ) ],
  [ rec( order := 3, degree := 3, baseSize := 1 ),
  <...many lines of output omitted...>
```

As one expects, the answer is computed in parallel in a time that decreases linearly with the number of processors. Hence a Beowulf cluster of 16 processors (1 ParGAP master and 15 ParGAP slaves) will return this information approximately 15 times as fast as a single processor, greatly facilitating interactive exploration of the properties of various groups.

If each single case can be executed quickly, then the execution time for `ParList()` can be dominated by the network overhead imposed by message passing between the master and slaves. In this case, an optional third parameter to `ParList()` specifies the number of cases to be submitted to a slave in a single message (modulo how many cases are left, of course). As an example, suppose we are interested in finding those of the first 100,000 integers that have many distinct prime factors, but to reduce the network overhead we wish to batch 1000 cases at a time to the slaves. Then one is able to execute the following.

```
gap> ParList([1..100000],
>            i -> Length(PrimePowersInt(i))/2, 1000);
```

4.2 Using ParGAP interactively

Having seen an example where ParGAP can be useful, one next needs to know what is involved in setting up and using ParGAP. ParGAP runs only on UNIX, due to its dependence on an MPI subset (included in the distribution) that calls UNIX sockets. It should be possible to port ParGAP to Windows, either through the use of Cygwin (UNIX emulation on Windows) or through a native Windows MPI implementation. Similarly, the code has been written in a "vanilla" manner, so that in principle, it should be easy to port to another interactive language, such as MAGMA or LISP. An older version was in fact written in LISP [Coo95], although the current version is not implemented in LISP. In the following, recall that ParGAP employs a master-slave architecture, with the local process being the "master" process, and all others being "slaves".

ParGAP is installed in the same way as other GAP packages. After extracting the ParGAP files, one invokes `configure` and `make`. (See Section 1.2 for the details.) One next writes a file with a list of slave processors to attach, and an indication of where ParGAP is located on that processor; the usual way is to name this file `procgroup` and have it in the directory one starts up ParGAP (see Section 1.3 for details). An example `procgroup` file is:

```
# This is a sample ParGAP procgroup file.
# Lines starting with a # are comments.
# Blank lines are also treated as comments.

# The following line generates the MASTER process
local 0
```

```
# Each of the following 'localhost' lines generates
# a SLAVE process on the same machine as the MASTER
localhost 1 /proj/sparc/gap4r3/pkg/pargap/bin/pargap.sh
localhost 1 /proj/sparc/gap4r3/pkg/pargap/bin/pargap.sh

# The following line generates a SLAVE process on a remote
# machine: 'iron.ccs.neu.edu' (called via rsh or ssh or ...)
iron.ccs.neu.edu 1 /proj/alpha/gap4r3/pkg/pargap/bin/pargap.sh
```

Next, instead of invoking `gap`, one then invokes the supplied shell script, `pargap.sh`, (possibly modified and/or renamed; again see Section 1.3 for the details). Observe that unlike most other GAP packages, one should not call `RequirePackage()` function from within a GAP session to start ParGAP. As usual, master and slave processes read a `.gaprc` file in the home directory, if present. ParGAP looks for a file named `procgroup` file in the current directory, but can be directed to use a different file through the `-p4pg` switch.

The following example assumes that `pargap` is a symbolic link or alias for the full pathname of `pargap.sh`. Hence, if one always uses ParGAP with the slave processes specified in `myprocgroup.big`, one may wish to set up an alias or else a symbolic link to a script containing the equivalent of:

`pargap.sh -p4pg` *PATH*`/myprocgroup.big`

Observe also that heterogeneous computing is supported, in that each slave machine may be a different dialect of UNIX running under different hardware.

Within ParGAP, the standard message-passing commands (send, receive, probe, etc.) are available. The slaves are numbered consecutively starting at 1. If no slave parameter is supplied for `SendMsg()` or `SendRecvMsg()`, then the default slave is 1. Most other commands take a default slave parameter, `MPI_ANY_SOURCE` (meaning "`ANY_SLAVE`"). `PingSlave()` sends a "null message", and waits for an acknowledgement that the slave is alive.

The command `SendMsg(`*string*`, `*i*`);` sends the (string) message *string* to slave i. Slave i then evaluates *string* as a GAP command, and returns an answer. If a command has no answer, (such as `"x:=3;"`), then the string "*no_return_val*" is returned. If multiple commands are sent, such as `"x:=3; y:=4; x+y;"`, then only the last return value is returned. Note that the final semicolon, normally required by GAP, is optional in a command string for a message.

Most of the other commands are clear from context. Every `SendMsg()` to a slave must be balanced by a `RecvMsg()`, except where the commands `FlushAllMsgs()` and `ParReset()` are used to flush any pending messages from a slave. A `SendRecvMsg(`*string*`, `*i*`)` command is equivalent to the consecutive commands `SendMsg(`*string*`, `*i*`)` and `RecvMsg(`*i*`)`. The command `BroadcastMsg()` sends its string argument to each slave, and there is no reply message. `ParEval()` is like `BroadcastMsg()`, but it also evaluates on the master process. If one is unsure whether there is a pending message, one can invoke `ProbeMsgNonBlocking()` or `ProbeMsg()`; if no slave parameter is provided, the default is `MPI_ANY_SOURCE`. The command `ParRead()` is a parallel version of GAP's `Read()` command; it is useful for ensuring that the slaves and master share a similar environment. Other examples of commands that are analogous to GAP's built-in commands include: `ParReread`, `ParEval`, `ParInstallGlobalFunction` and `ParInstallGlobalFunction`. See Section 2.1 for more complete descriptions of the above commands.

An illustration of the usage of the commands mentioned above with a sample ParGAP session, for which a `procgroup` file has defined two slaves is the first example given in Section 1.4. It's recommended that you review that example at this point.

4.3 Streaming

Next, we demonstrate a simple implementation of "streaming" as an example of how easy it is to use the ParGAP tools to provide new functionality. To the best of this author's knowledge, the term "streaming" originated with Charles Leedham-Green at the conference on which this proceedings is based, where he made a general request for streaming functionality to be implemented in some software system.

The idea of streaming is that one may have two algorithms for solving a problem. One would like to to solve the problem simultaneously on two processors, each using a distinct algorithm. Whichever processor finds the solution first should report back to the master process. The master process should then interrupt the other slave, so as to make it available for further computation. This is useful when one has a heuristic available that may finish early with the correct answer, or may continue for a very long time. The heuristic can then be "streamed" alongside the standard algorithm, in full confidence that the standard algorithm will provide a reasonable upper bound on the time to compute an answer.

One way to provide "streaming" functionality is by the implementation of a function we describe below that is inspired by GAP's `First()` function (see "ref:first" in the GAP Reference Manual). The function `First()` takes a list and boolean function as arguments, and returns the first element of the list for which the boolean function returns `true`. In contrast, we define a corresponding ParGAP function, `ParFirstResult()`, which takes a list and a (general) function as arguments. Messages are sent to the slaves causing the ith slave to evaluate the function on the ith list element. The value returned is the value obtained from whichever slave finishes first. Note that in consistency with the goal of streaming, the function signals an error if asked to evaluate a list with more entries than the number of slaves.

```
gap> ParFirstResult := function( list, fnc )
>   local i, result;
>   if Length(list) > TOPCnumSlaves then
>     Error("too few slaves");
>   fi;
>   for i in [1..Length(list)] do
>     SendMsg( PrintToString(
>                   "fnc :=", fnc, "; fnc(", list[i], ");"),
>              i );
>   od;
>   result := RecvMsg(); # default is MPI_ANY_SOURCE
>   ParReset(); # Interrupt all other slaves
>   return result;
> end;;
```

We now demonstrate the use of `ParFirstResult()` in "streaming". For our example we need the FactInt package by Stefan Kohl. To get this package if you don't have it, visit

`http://www-gap.dcs.st-and.ac.uk/gap/Share/factint.html`

or the equivalent at one of GAP's mirror sites, and follow the easy installation instructions.

If one hasn't already included a `RequirePackage("factint");` statement in one's `.gaprc` file then it is necessary to do:

```
gap> ParEval("RequirePackage(\"factint\")");

Loading FactInt 1.1 (Routines for Integer Factorization),
by Stefan.Kohl@cip.mathematik.uni-stuttgart.de

true
```

so that each slave (not just the master) is aware of the FactInt functions.

Above we defined `ParFirstResult()` on the master process. We will assume that we have two slaves.

```
gap> StreamingFactInt := function(i, x)
>   local alg;
>   alg :=
>    [ x -> [ "MPQS ALGORITHM",  FactorsMPQS(Factorial(x)+1) ],
>      x -> [ "CFRAC ALGORITHM", FactorsCFRAC(Factorial(x)+1) ]
>    ];
>   return alg[i](x);
> end;;
```

Both `StreamingFactInt(1, x);` and `StreamingFactInt(2, x);` factor an integer, but one uses the multiple polynomial quadratic sieve algorithm (MPQS), and the other uses the continued fraction algorithm (CFRAC); the functions `FactorsMPQS` and `FactorsCFRAC` that perform these algorithms are defined by the FactInt package. We demonstrate the "streaming" of these algorithms in determining the factorization of $35! + 1$.

```
gap> # Now, define StreamingFactInt() on each of the slaves.
gap> BroadcastMsg( PrintToString( "StreamingFactInt :=",
>                                 StreamingFactInt ) );
gap> ParFirstResult([1,2], i->StreamingFactInt(i, 35) );
... resetting ...
[ "MPQS ALGORITHM", [ 137, 379, 17839, 340825649, 32731815563800396289317 ] ]
```

4.4 TOP-C model for non-trivial parallelism

There are many examples where trivial parallelism does not suffice. Typically, this happens either when there are global variables that must be "read" by each slave process, and possibly updated, or else when the input for the next slave process depends on the result that arrived from the last slave process. Here we provide only the basics of the parallel model. The parallel model was described in more detail in [Coo97], and a still more detailed description is contained in Chapter 3.

For non-trivial parallelism, ParGAP uses the TOP-C model [Coo96]. ParGAP typically invokes the TOP-C model via a command such as

`MasterSlave(` *GenerateTaskInput*`,` *DoTask*`,` *CheckTaskResult*`,`
UpdateSharedData `);`

The four arguments, *GenerateTaskInput*, *DoTask*, *CheckTaskResult*, and *UpdateSharedData* are "callback" functions written by the application programmer, and the names of those callback functions are arbitrary. (The manual for ParGAP/MPI, the earlier incarnation of ParGAP used slightly different names for its examples, but the purpose of the arguments of `MasterSlave()` has not changed – only the naming convention has.)

The only ParGAP command above is `MasterSlave()`. A task is an arbitrary function (here called *DoTask*`()`) executed on a slave process, that takes input from the master process and returns its output to the master process. The diagram in Section 3.2 gives some idea of the flow of control as a task is processed. The diagram there is meant to represent the three main abstractions of the TOP-C model: (1) the task, (2) the shared data, and (3) the action. The **shared data** consists of any globally shared data (which should be readable by all processes). The **task** is a procedure executed on a slave process that takes a **task input**, and the shared data, and produces a **task output**, which depends only on the task input and shared data. Finally, the task input and task output are sent to the master process, which must then decide upon an **action**.

Typical actions are `NO_ACTION` (and the master process might save the task output in a private list of results), `UPDATE_SHARED_DATA` (send a message to all slaves to update the local copy of the globally shared data), and `REDO` (re-do the computation in case the shared data was changed by another slave process). (Earlier

incarnations of ParGAP used UPDATE. Now the alternative UPDATE_SHARED_DATA is offered and we standardize on this here.)

An example invocation of MasterSlave() is shown below, where we pass the four application functions as direct arguments of MasterSlave(). The routine below implements a simplified version of the ParList() function described in Section 4.1.

```
ParInstallTOPCGlobalFunction( "MyParList",
function( list, fnc )
  local result, iter;
  result := [];
  # Each invocation of GAP's iterator
  # returns next element of list.
  iter := Iterator(list);
  MasterSlave(
      # GenerateTaskInput():
      function() if IsDoneIterator(iter) then return NOTASK;
                 else return NextIterator(iter); fi; end,
      # DoTask():
      fnc,
      # CheckTaskResult():
      function(input,output) result[input] := output;
                             return NO_ACTION; end,
      # UpdateSharedData():
      Error # We never see action:  UPDATE_SHARED_DATA
      );
  return result;
end );
```

The function ParInstallTOPCGlobalFunction() installs MyParList on all ParGAP processes. It also defines the version of MyParList() on the master differently from on a slave, so that a call, MyParList([1..100], x->x^2);, on the master automatically causes MyParList([1..100], x->x^2); to be invoked on the slave with the same arguments. This is required for the TOP-C model. Note that the distinct function, ParInstallGlobalFunction(), exists for the equivalent of BroadcastMsg("InstallGlobalFunction(MyParList, function ... end)");. Both functions should be called only from the master (possibly from inside a Read() command).

The shared data consists only of the variable fnc, which is read by all processes, but in this case is never "updated". Note that one need not explicitly declare variables that are in the shared data. A TOP-C shared data variable is defined as a variable whose value is read by more than one process, and which is modified only through a call to the application routine *UpdateSharedData*(). If we would like to see the messages, as they are passed back and forth between master and slave, we can optionally set the variable ParTrace (see 6.1.1). We are then ready to execute our new parallel function.

```
gap>  ParTrace := true;;
gap>  MyParList([2..256], AnalyzePrimitivePermGroupsOfOrder);
```

The example above in fact requires only trivial parallelism. Hence the *CheckTaskResult*() parameter to MasterSlave() is especially simple. Since the action is never REDO, we achieve the maximum concurrency among slave processes, resulting in a speedup that is linear in the number of slaves.

In general, the parallel efficiency of an application, with its concommitant decisions about concurrency of slave tasks are typically determined by the actions chosen by *CheckTaskResult*(). The sample below is a standard ParGAP "idiom" that allows one to easily set up reasonable concurrency in a parallel application.

```
CheckTaskResult := function( taskInput, taskOutput )
  if taskOutput = fail then return NO_ACTION;
  elif not IsUpToDate() then return REDO_ACTION;
  else return UPDATE_ACTION;
  fi;
end;
```

The function `IsUpToDate()` is a boolean function provided by ParGAP that returns `true` if there has been no `UPDATE_SHARED_DATA` action since the time that the "corresponding" task input was generated by a call to the user-provided function *GenerateTaskInput*`()`. Otherwise, `IsUpToDate()` returns `false`. The "corresponding" task input is the task input associated with that task output which was most recently seen on the master process. In the context of the user-provided function *CheckTaskResult*`()`, it is the first argument to that function.

5 MasterSlave Tutorial

This chapter assumes the background knowledge in section 3.1. ParGAP must be invoked through a script like `pargap.sh` generated in ParGAP's `bin` during installation. In addition, there must be a `procgroup` file in the current directory when ParGAP is called, or alternatively, `-p4pg` *FULL_PATH*`/procgroup` may be included on the `pargap.sh` command line. See Section 1.3 for the details. You should find a sample `procgroup` file in ParGAP's `bin` directory after installation of ParGAP. See the section 1.6 for further details. Many of the examples of this section can be found in ParGAP's `examples` directory.

5.1 A simple example

A master-slave computation is invoked when a ParGAP program issues the command `MasterSlave()`. This command is an example of what is called "collective communication" in MPI (although the command is not part of MPI). It is also sometimes called SPMD (Single Program, Multiple Data), since all processes see the same code, although different processes may execute different parts of the code. The `MasterSlave()` command must be invoked on all processes before execution can begin. The following trivial example does this. (Note that the final `\` on a line that is still inside a string allows continuation of a string to the next line.) We illustrate these principles first in their simplest form, making all variables global variables. Later, we introduce additional ParGAP utilities that allow one to write in better style.

```
#Shared Data: none
#TaskInput:   counter
#TaskOutput:  counter^2 (square of counter)
#Task:        compute counter^2 from counter
ParEval( "result := []" );
ParEval( "counter := 0; \
          SubmitTaskInput := function() \
            counter := counter + 1; \
            if counter <= 10 then return counter; else return NOTASK; fi; \
          end;");
ParEval( "DoTask := x->x^2" );
ParEval( "CheckTaskResultVers1 := function( input, output ) \
            result[input] := [input, output]; \
            return NO_ACTION; \
          end;" );
ParEval( "MasterSlave( SubmitTaskInput, DoTask, CheckTaskResultVers1 )" );
Print(result);
```

By default, `ParTrace = true` (see 6.1.1), causing the execution to display each input, `x`, as it is passed from the master to a slave, and each output, `x^2`, as it is passed from the slave back to the master. This behavior can be turned off by setting: `ParTrace := false;` The fourth argument of `MasterSlave()`, `Print`, is a dummy argument that is never invoked in this example.

Note that the result list is filled in only on the local process, and was never defined or modified on the slave processes. To remedy this situation, we introduce the concept of **shared data**, a globally shared, application-defined data structure. A central principle of the TOP-C model in ParGAP is that any routine may "read" the

shared data, but it may be modified only by the application-defined routine, *UpdateSharedData*(). Hence, if we wanted the result list to be recorded on all processes (perhaps as a lookup table), we would now write:

```
ParBroadcast(PrintToString("result = ", result));
```

5.2 ParSquare

Until now, we have been using global variables. It is better style to use local variables, where possible. We rewrite the above routine in the improved style:

```
#Shared Data: result [ result is shared among all processes ]
#TaskInput:   counter
#TaskOutput:  counter^2 (square of counter)
#Task:        compute counter^2 from counter
#UpdateSharedData:  record [counter, counter^2] at index, counter, of result
MSSquare := function( range ) # expects as input a range, like [1..10]
  local counter, result,
      SubmitTaskInput, DoTask, CheckTaskResultVers2, UpdateSharedData;
  counter := range[1]; # Reset counter for use in SubmitTaskInput()
  result := [];
  SubmitTaskInput := function()
    counter := counter + 1;
    if counter <= range[Length(range)] then return counter;
    else return NOTASK;
    fi;
  end;
  DoTask := x->x^2;
  CheckTaskResultVers2 := function( input, output )
    return UPDATE_ACTION;
  end;
  UpdateSharedData := function( input, output )
    result[input] := [input, output];
  end;
  MasterSlave( SubmitTaskInput, DoTask, CheckTaskResultVers2, UpdateSharedData );
  return result;
end;

#ParSquare() is the main calling function;  It must define MSSquare on
#  all slaves before calling it in parallel.
ParSquare := function( range ) # expects as input a range, like [1..10]
  ParEval( PrintToString( "MSSquare := ", MSSquare ) );
  return ParEval( PrintToString( "MSSquare( ", range, ")" ) );
end;
```

5.3 ParInstallTOPCGlobalFunction() and TaskInputIterator() (ParSquare revisited)

This example can be written more compactly by using some of the convenience functions provided by ParGAP. Specifically, we would rewrite this as:

```
ParInstallTOPCGlobalFunction( "ParSquare", function( range )
  local result;
  result := [];
  MasterSlave( TaskInputIterator( range ),
```

```
                x->x^2,
                function( input, output ) return UPDATE_ACTION; end,
                function( input, output ) result[input] := [input, output]; end
              );
  return result;
end );
```

The usage above demonstrates the use of two utilities.

1 ▶ `ParInstallTOPCGlobalFunction(` *string*, *function* `)` F

This defines *gvar* as a function on the master and on the slaves. On each slave, the definition of *gvar* is given by *function*. However, on the master, *gvar* is defined as a function that first calls *gvar* on all slaves with the arguments originally passed to *gvar*, and then on the master, *function* is called with the original arguments. This is exactly the behavior that is wanted in order to compress an invocation of `MasterSlave()` so that the right things happen on both the master and on the slaves. This is exactly what we saw in the previous definition of `ParSquare`, above.

2 ▶ `TaskInputIterator(` *collection* `)` F

This function provides the functionality of a common case of `SubmitTaskInput()`, by turning it into a GAP iterator (see "ref:iterators" in the Reference Manual). Its meaning is best understood from its definition:

```
TaskInputIterator := function( collection )
  local iter;
  iter := Iterator( collection );
  return function()
           if IsDoneIterator(iter) then return NOTASK;
           else return NextIterator(iter);
           fi;
         end;
end;
```

5.4 ParMultMat

Let us now write a matrix-matrix multiplication routine in this style. Since matrix multiplication for dimension n requires n^3 operations, we can afford to spend n^2 time doing any sequential work. (A finer analysis would also consider the number of slaves, k, resulting in up to $k * n^2$ time to send all messages, depending on the MPI broadcast algorithm.) So, a sequential matrix multiplication program might be written as follows. (The style emphasizes clarity over efficiency.)

```
SeqMultMat := function(m1, m2)               # sequential code
  local i, j, k, n, m2t, sum, result;
  n := Length(m1);
  result := [];
  m2t := TransposedMat(m2);
  for i in [1..n] do
    result[i] := [];
    for j in [1..n] do
      sum := 0;
      for k in [1..n] do
        sum := sum + m1[i][k]*m2t[j][k];
      od;
      result[i][j] := sum;
    od;
```

```
  od;
  return result;
end;
```

We choose to define the task as the computation of a single row of the result matrix. This corresponds to the body of the outermost loop.

```
#Shared Data: m1, m2t, result (three matrices)
#TaskInput:   i (row index of result matrix)
#TaskOutput:  result[i] (row of result matrix)
#Task:        Compute result[i] from i, m1, and m2
#UpdateSharedData:  Given result[i] and i, modify result on all processes.

ParInstallTOPCGlobalFunction( "ParMultMat", function(m1, m2)
  local i, n, m2t, result, DoTask, CheckTaskResult, UpdateSharedData;
  n := Length(m1);
  result := [];
  m2t := TransposedMat(m2);

  DoTask := function(i) # i is task input
    local j, k, sum;
    result[i] := [];
    for j in [1..n] do
      sum := 0;
      for k in [1..n] do
        sum := sum + m1[i][k]*m2t[j][k];
      od;
      result[i][j] := sum;
    od;
    return result[i]; # return task output, row_i
  end;
  # CheckTaskResult executes only on the master
  CheckTaskResult := function(i, row_i) # task output is row_i
    return UPDATE_ACTION; # Pass on output and input to UpdateSharedData
  end;
  # UpdateSharedData executes on the master and on all slaves
  UpdateSharedData := function(i, row_i) # task output is row_i
    result[i] := row_i;
  end;
  # We're done defining the task.  Let's do it now.
  MasterSlave( TaskInputIterator([1..n]), DoTask, CheckTaskResult,
               UpdateSharedData );
  # result is defined on all processes;  return local copy of result
  return result;
end );
```

5.5 DefaultCheckTaskResult (as illustrated by ParSemiEchelonMatrix)

Now that the basic principles of the TOP-C model are clear, we investigate an example that requires most of the basic features of ParGAP, including the use of `IsUpToDate()` and `REDO_ACTION`. Recall the standard idiom for *CheckTaskResult*`()`. These issues were discussed in the section 3.4.

```
DefaultCheckTaskResult := function( taskOutput, taskInput )
  if taskOutput = false then return NO_ACTION;
  elif not IsUpToDate() then return REDO_ACTION;
  else return UPDATE_ACTION;
  fi;
end;
```

The next example is a parallelization of the function `SemiEchelonMat()` (a form of Gaussian elimination) in the GAP library, `lib/matrix.gi`. Unlike the previous examples, parallelizing Gaussian elimination efficiently is a non-trivial undertaking. This is because a naive parallelization has poor load balancing. A slave executing a task in the middle will have to *sift* a row vector through many previous row vectors, while a slave executing a task toward the beginning or end will have little work to do. We will begin with a naive parallelization based on the sequential code, and then migrate the code in a natural manner toward a more efficient form, by analyzing the inefficiencies and applying the TOP-C model.

The reader may wish to stop and read the original code in `lib/matrix.gi` first. The logic of `SemiEchelonMat()` is to examine each row vector of an input matrix, in order, reduce it by a list of basis vectors stored in `vectors`, and then add the row to `vectors`. Upon completion, the number of leading zeroes of the row vectors in `vectors` may not increase monotonically, but each element of `vectors` will have a unique number of leading zeroes. Some rows of the input matrix may reduce to the zero matrix, in which case they are not added to `vectors`.

For the reader's convenience, the original sequential code is reproduced here.

```
SemiEchelonMat := function( mat )

    local zero,      # zero of the field of <mat>
          nrows,     # number of rows in <mat>
          ncols,     # number of columns in <mat>
          vectors,   # list of basis vectors
          heads,     # list of pivot positions in 'vectors'
          i,         # loop over rows
          j,         # loop over columns
          x,         # a current element
          nzheads,   # list of non-zero heads
          row;       # the row of current interest

    mat:= List( mat, ShallowCopy );
    nrows:= Length( mat );
    ncols:= Length( mat[1] );

    zero:= Zero( mat[1][1] );

    heads:= ListWithIdenticalEntries( ncols, 0 );
    nzheads := [];
    vectors := [];

    for i in [ 1 .. nrows ] do
        row := mat[i];
        # Reduce the row with the known basis vectors.
        for j in [ 1 .. Length(nzheads) ] do
            x := row[nzheads[j]];
            if x <> zero then
                AddRowVector( row, vectors[ j ], - x );
```

```
            fi;
        od;
        j := PositionNot( row, zero );
        if j <= ncols then
            # We found a new basis vector.
            MultRowVector(row, Inverse(row[j]));
            Add( vectors, row );
            Add( nzheads, j);
            heads[j]:= Length( vectors );
        fi;
    od;

    return rec( heads   := heads,
                vectors := vectors );
    end;
```

Although GAP's Gaussian elimination algorithm appears to be awkward to parallelize (since the next row vector in `vectors` depends on row reduction by all previous vectors, we will see how the original code of `SemiEchelonMat()` can be modified in a natural manner to produce useful parallelism. This illustrates the general TOP-C paradigm for "naturally" parallelizing a sequential algorithm.

```
#Shared Data: vectors (basis vectors), heads, nzheads, mat (matrix)
#TaskInput:   i (row index of matrix)
#TaskOutput:  List of (1) j and (2) row i of matrix, mat, reduced by vectors
#               j is the first non-zero element of row i
#Task:        Compute reduced row i from mat, vectors, heads
#UpdateSharedData:  Given i, j, reduced row i, add new basis vector
#               to vectors and update heads[j] to point to it

ParInstallTOPCGlobalFunction( "ParSemiEchelonMat", function( mat )
    local zero,      # zero of the field of <mat>
          nrows,     # number of rows in <mat>
          ncols,     # number of columns in <mat>
          vectors,   # list of basis vectors
          heads,     # list of pivot positions in 'vectors'
          i,         # loop over rows
          nzheads,   # list of non-zero heads
          DoTask, UpdateSharedData;

    mat:= List( mat, ShallowCopy );
    nrows:= Length( mat );
    ncols:= Length( mat[1] );

    zero:= Zero( mat[1][1] );

    heads:= ListWithIdenticalEntries( ncols, 0 );
    nzheads := [];
    vectors := [];
```

```
    DoTask := function( i ) # taskInput = i
      local j,         # loop over columns
            x,         # a current element
            row;       # the row of current interest
      row := mat[i];
      # Reduce the row with the known basis vectors.
      for j in [ 1 .. Length(nzheads) ] do
          x := row[nzheads[j]];
          if x <> zero then
              AddRowVector( row, vectors[ j ], - x );
          fi;
      od;
      j := PositionNot( row, zero );
      if j <= ncols then return [j, row]; # return taskOutput
      else return fail; fi;
    end;
    UpdateSharedData := function( i, taskOutput )
      local j, row;
      j := taskOutput[1];
      row := taskOutput[2];
      # We found a new basis vector.
      MultRowVector(row, Inverse(row[j]));
      Add( vectors, row );
      Add( nzheads, j);
      heads[j]:= Length( vectors );
    end;

    MasterSlave( TaskInputIterator( [1..nrows] ), DoTask, DefaultCheckTaskResult,
                  UpdateSharedData );

    return rec( heads   := heads,
                vectors := vectors );
end );
```

The next section describes how to make this code more efficient.

5.6 Caching slave task outputs (ParSemiEchelonMat revisited)

The code above is inefficient unless `nrows >> ncols`. This is because if `nrows` is comparable to `ncols`, it will be rare for `DoTask()` to return `fail`. If most slaves return a result distinct from `fail`, then `DefaultCheckTaskResult()` will return an `UPDATE_ACTION` upon receiving the output from the first slave, and it will return a `REDO_ACTION` to all other slaves, until those slaves execute `UpdateSharedData()`. The inefficiency arose because a `REDO_ACTION` caused the original slave process to re-compute `DoTask()` from the beginning.

In the case of a `REDO_ACTION`, we can fix this by taking advantage of information that was already computed. To accomplish this, a global variable should be defined on all slaves:

```
ParEval("globalTaskOutput := [[-1]]");
```

the routine `DoTask()` in the previous example should be modified to:

```
DoTask := function( i )
    local j,         # loop over columns
          x,         # a current element
          row;       # the row of current interest
  if i = globalTaskOutput[1] then
    # then this is a REDO_ACTION
    row := globalTaskOutput[2]; # recover last row value
  else row := mat[i];
  fi;
  # Reduce the row with the known basis vectors.
  for j in [ 1 .. Length(nzheads) ] do
    x := row[nzheads[j]];
    if x <> zero then
      AddRowVector( row, vectors[ j ], - x );
    fi;
  od;
  j := PositionNot( row, zero );
  # save row in case of new REDO_ACTION
  globalTaskOutupt[1] := i;
  globalTaskOutput[2] := row;
  if j <= ncols then return [j, row]; # return taskOutput
  else return fail; fi;
end;
```

(A perceptive reader will have noticed that it was not necessary to also save and restore `row` from `globalTaskOutput`, since this can be found again based on the saved variable value `i`. However, the additional cost is small, and it illustrates potentially greater generality of the method.)

The next section describes how to make this code more efficient.

5.7 Agglomerating tasks for efficiency (ParSemiEchelonMat revisited again)

A more efficient parallelization would partition the matrix into sets of adjacent rows, and send an entire set as a single `taskInput`. This would minimize the communication overhead, since the network latency varies only slowly with message sizxe, but linearly with the number of messages. To minimize network latency, one adds an extra parameter to `MasterSlave()` in order to bundle, perhaps, up to 5 tasks at a time.

```
MasterSlave( TaskInputIterator( [1..n] ), DoTask, DefaultCheckTaskResult,
             UpdateSharedData, 5 );
```

Now the task input will be a list of the next 5 tasks returned by `GetTaskInput()`, or in this case by `TaskInputIterator( [1..nrows] )`. If fewer than 5 tasks are produced before `NOTASK` is returned, then the task input will be correspondingly shorter. If the first input task is `NOTASK` (yielding a list of tasks of length 0), then this will be interpreted as a traditional `NOTASK`. The task output corresponding to this task input is whatever the application routine, `DoTask()` produces as task output. The routine `DoTask()` will be unchanged, and `MasterSlave()` will arrange to repeatedly call `DoTask()`, once for each input task and produce a list of task outputs.

Hence, this new variation requires us to rewrite `UpdateSharedData()` in the obvious manner, to handle a list of input and output tasks. Here is one solution to patch the earlier code.

1 ▶ `TaskAgglomIndex` V

This global variable is provided for use inside `DoTask()`. It allows the application code to inquire about the index of the input task in the full list of tasks created when agglomTask is used. The variable is most useful in the case of a `REDO_ACTION` or `CONTINUATION_ACTION()`, as illustrated below.

```
ParEval("globalTaskOutput := [[-1]]");
ParEval("globalTaskOutputs := []");

#Shared Data: vectors (basis vectors), heads, mat (matrix)
#TaskInput:   i (row index of matrix)
#TaskOutput:  List of (1) j and (2) row i of matrix, mat, reduced by vectors
#               j is the first non-zero element of row i
#Task:        Compute reduced row i from mat, vectors, heads
#UpdateSharedData:  Given i, j, reduced row i, add new basis vector
#               to vectors and update heads[j] to point to it

ParInstallTOPCGlobalFunction( "ParSemiEchelonMat", function( mat )
  local zero,      # zero of the field of <mat>
        nrows,     # number of rows in <mat>
        ncols,     # number of columns in <mat>
        vectors,   # list of basis vectors
        heads,     # list of pivot positions in 'vectors'
        i,         # loop over rows
        nzheads,   # list of non-zero heads
        DoTask, UpdateSharedDataWithAgglom;

  mat:= List( mat, ShallowCopy );
  nrows:= Length( mat );
  ncols:= Length( mat[1] );

  zero:= Zero( mat[1][1] );

  heads:= ListWithIdenticalEntries( ncols, 0 );
  nzheads := [];
  vectors := [];

  DoTask := function( i )
      local j,         # loop over columns
            x,         # a current element
            row;       # the row of current interest
    if IsBound(globalTaskOutputs[TaskAgglomIndex])
       and i = globalTaskOutputs[TaskAgglomIndex][1] then
      # then this is a REDO_ACTION
      row := globalTaskOutputs[TaskAgglomIndex][2][2]; # recover last row value
    else row := mat[i];
    fi;
    # Reduce the row with the known basis vectors.
    for j in [ 1 .. Length(nzheads) ] do
      x := row[nzheads[j]];
      if x <> zero then
        AddRowVector( row, vectors[ j ], - x );
      fi;
```

```
    od;
    j := PositionNot( row, zero );

    # save [input, output] in case of new REDO_ACTION
    globalTaskOutputs[TaskAgglomIndex] := [ i, [j, row] ];

    if j <= ncols then return [j, row]; # return taskOutput
    else return fail; fi;
  end;

  # This version of UpdateSharedData() expects a list of taskOutput's
  UpdateSharedDataWithAgglom := function( listI, taskOutputs )
    local j, row, idx, tmp;
    for idx in [1..Length( taskOutputs )] do
      j := taskOutputs[idx][1];
      row := taskOutputs[idx][2];

      if idx > 1 then
        globalTaskOutputs[1] := [-1, [j, row] ];
        tmp := DoTask( -1 ); # Trick DoTask() into a REDO_ACTION
        if tmp <> fail then
          j := tmp[1];
          row := tmp[2];
        fi;
      fi;

      # We found a new basis vector.
      MultRowVector(row, Inverse(row[j]));
      Add( vectors, row );
      Add( nzheads, j);
      heads[j]:= Length( vectors );
    od;
  end;

  MasterSlave( TaskInputIterator( [1..nrows] ), DoTask, DefaultCheckTaskResult,
               UpdateSharedDataWithAgglom, 5 ); #taskAgglom set to 5 tasks

  return rec( heads   := heads,
              vectors := vectors );
end );
```

Note that in this simple example, we were able to re-use most of the code from the previous version, at the cost of adding an additional global variable, `globalTaskOutputs`. In fact, the last `DoTask()` is backward compatible to the first version of the code, for which `agglomTasks` is not used. If we wanted to run the latest code without agglomeration of tasks, it would suffice either to set the `taskAgglom` parameter to 1, or else to remove it entirely and replace `UpdateSharedDataWithAgglom()` by `UpdateSharedData()`.

It is useful to experiment with the above code by substituting a variable, `taskAgglom`, for the number 5, and trying it out with remote slaves on your own network for different values of `taskAgglom` and for different size matrices. You can call `MasterSlaveStats()` to see the effect of different parameters. Suitable pseudo-random matrices can be quickly generated via `mat := PseudoRandom( GL( 30, 5 )` and similar commands.

The paper [Coo98] is suggested as further reading to see a still more efficient parallel implementation of `ParSemiEchelonMatrix` (also known as Gaussian elimination) using the TOP-C model.

5.8 Raw MasterSlave (ParMultMat revisited)

Finally, we given an example of a variation of `MasterSlave()`, based on a "raw" `MasterSlave()`. These versions are designed for the common case of legacy code that contains deeply nested parentheses. The `taskInput` may be generated inside several nested loops, making it awkward and error-prone to produce a function, `SubmitTaskInput()`, that will generate instances of `taskInput` in the appropriate sequence.

Effectively, when we wish to return successive values from several deeply nested loops, we are in the situation of programming the "opposite of a GAP iterator" (see "ref:iterators" in the Reference Manual). We are already producing successive iterator values, and we wish to "stuff them back into some iterator". Until GAP develops such a language construct `:-)` , the following example of a "raws" `MasterSlave()` demonstrates a solution. Before studying this example, please review the sequential version, `SeqMultMat()` near the beginning of section 5.4.

We make use of the following three new ParGAP functions.

1 ▶ `BeginRawMasterSlave(` *DoTask*`,` *CheckTaskResult*`,` *UpdateSharedData* `)` F
▶ `RawSubmitTaskInput(` *taskInput* `)` F
▶ `EndRawMasterSlave()` F

Their use will be obvious in the next example. This time, we parallelize `SeqMultMat()` by defining the task as the computation of a single entry in the result matrix. Hence, the task will be the computation of the appropriate inner product. For dimension n, n^2 tasks are now generated, and each task is generated inside a doubly nested loop.

```
#Shared Data: m1, m2t, result (three matrices)
#TaskInput:   [i,j] (indices of entry in result matrix)
#TaskOutput:  result[i][j] (value of entry in result matrix)
#Task:        Compute inner produce of row i of m1 by colum j of m1
              ( Note that column j of m1 is also row j of m2t, the transpose )
#UpdateSharedData:  Given result[i][j] and [i,j], modify result everywhere

ParInstallTOPCGlobalFunction( "ParRawMultMat", function(m1, m2)
  local i, j, k, n, m2t, sum, result, DoTask, CheckTaskResult, UpdateSharedData;
  n := Length(m1);
  m2t := TransposedMat(m2);
  result := ListWithIdenticalEntries( Length(m2t), [] );

  DoTask := function( input )
    local i,j,k,sum;
    i:=input[1]; j:=input[2];
    sum := 0;
    for k in [1..n] do
      sum := sum + m1[i][k]*m2t[j][k];
    od;
    return sum;
  end;

  CheckTaskResult := function( input, output )
    return UPDATE_ACTION;
  end;
```

```
  UpdateSharedData := function( input, output )
    local i, j;
    i := input[1]; j := input[2];
    result[i][j] := output;
    # result[i,j] := sum;
  end;

  BeginRawMasterSlave( DoTask, CheckTaskResult, UpdateSharedData );
  for i in [1..n] do
    result[i] := [];
    for j in [1..n] do
      RawSubmitTaskInput( [i,j] );
      # sum := 0;
      # for k in [1..n] do
      #   sum := sum + m1[i][k]*m2t[j][k];
      # od;
      # result[i][j] := sum;
    od;
  od;
  EndRawMasterSlave();

  return result;
end );
```

6 Advanced Concepts for TOP-C model (MasterSlave)

This chapter may be safely skipped on a first reading. If you still want to read this chapter, it should mean that you are familiar with the basics of the TOP-C model, and are looking for advice on how to use the model more effectively. The first piece of advice is that the choice of **task** and **shared data** interact strongly with the choice of parallel algorithm. We review those concepts more precisely here, in light of the overall context of the TOP-C model.

task:
: A task is a function that that takes a single argument, *taskInput*, reads certain globally shared data, the **shared data**, and computes a result, the *taskOutput*. Hence, given the same task input and the same shared data, a task should always compute the same task output. The TOP-C model implements this concept through the *DoTask*`()` application routine. In the TOP-C model, this rule is bent to accommodate caching of private data to efficiently handle a `REDO_ACTION` (see Section 5.6), or to accommodate a `CONTINUATION_ACTION()` (see Section 3.6).

shared data:
: The **shared data** is globally shared data. It should be initialized before entering `MasterSlave()`. The shared data is never explicitly declared. However, it is best for the application programmer to include a comment specifying the shared data for his or her application. The TOP-C model poses certain restrictions on what legally constitutes shared data. The shared data must include enough of the global data (variables that occur **free** in the *DoTask*`()` procedure) so that the task output of *DoTask*`()` is uniquely determined by the task input and the shared data. However, the shared data must **not** include any variables whose values are modified outside of the application routine *UpdateSharedData*`()`. Also, the shared data is updated **non-preemptively**, in the sense that a slave process will always complete its current task before reading a newly arrived message that invokes *UpdateSharedData*`()`. If a slave privately caches data for purposes of a `REDO_ACTION` or `CONTINUATION_ACTION()`, such data is explicitly not part of the shared data.

6.1 Tracing and Debugging

In testing a program using `MasterSlave()`, a hierarchy of testing is suggested. The principle is to test the simplest example first, and then iterate to more complex examples. When a stable portion of the program is ready for testing, the following sequence of tests is suggested:

sequential
: Replace `MasterSlave()` by `SeqMasterSlave()` (see definition below) and see if the program performs correctly. `SeqMasterSlave()` will run only on the master, without sending any messages, and so the full range of sequential debugging tools is available.

one slave
: Restore `MasterSlave()` and set up the `procgroup` file to have only one slave process (one line, `local 0`, and one line `localhost ...`). Initially test with no `taskAgglom` parameter for `MasterSlave()`, and then test with the full set of parameters.

two slaves
: Same advice as for one slave, but two lines: `localhost ...`

many slaves
: Full scale test, both without and with `taskAgglom`.

1 ▶ `ParTrace` V

A second easy testing strategy is to set `ParTrace` to `true`. (This is the default value.) This causes all *taskInput*s, *taskOutput*s, and non-trivial actions (actions other than `NO_ACTION`) to be displayed at the terminal. The information is printed in the same sequence as seen by the master process.

Another "cheap" debugging trick is to inspect the values of global variables on the slave after it has been thrown out of the `MasterSlave()` procedure. The following code demonstrates by interrogating the sum of the variables `x` and `y` on slave number 2.

```
gap> SendRecvMsg("x+y;\n", 2);
```

This is useful to inspect cached data on a slave used for a `REDO_ACTION` or `CONTINUATION_ACTION()`. It may also be useful to verify if the shared data on the slave is the same as on the master. If the slave process is still inside the procedure `MasterSlave()`, then from within a break loop on the master, you may also want to interactively call *DoTask*`(` *testInput* `)` to determine if the expected *taskOutput* is produced.

If the master process is still within `MasterSlave()`, then it is useful to execute *DoTask*`()` locally on the master process, and debug this sequentially.

There is also the time-honored practice of inserting print statements. Print statements "work" both on the master and on the slaves. If ParTrace produces too much output, or not the right kind of information, one can add print statements exactly where one needs them. As with any UNIX debugging, it is sometimes useful to include a call to `fflush(stdout)` to force any pending output. ParGAP binds this to:

2 ▶ `UNIX_FflushStdout()` F

This has the same effect as the UNIX `fflush(stdout)`. There may be pending output in a buffer, that UNIX delays printing for efficiency. Printing any remaining output in the buffer is forced by this command. A common sequence is: `Print("information"); UNIX_FflushStdout();`. Note also that when the slave prints, there are "two" standard outputs involved. You may also want to include a call to `UNIX_FflushStdout()` on the master to force any pending output that originated on a slave. Finally, you should be conscious of network delays, and so a print statement in a slave process will typically take longer to appear than a print statement in the master process.

3 ▶ `SeqMasterSlave(` *SubmitTaskInput*`,` *DoTask*`[,` *CheckTaskResult*`[,`*UpdateSharedData*`[,` *taskAgglom* `]]]` `)` F

If a bug is exhibited even in the context of a single slave, then the code is "almost" sequential. In this case, one can test further by replacing the call to `MasterSlave()` by a call to `SeqMasterSlave()`, and debug in a context that involves zero messages and no interaction with any slave. It can also be helpful to carry out initial debugging in this context. Note that in the case of a single slave, which is what `SeqMasterSlave` emulates, `IsUpToDate()` will always return true, and so most applications will not call for a `REDO_ACTION`.

6.2 Efficiency Considerations

There are two common reasons for loss of efficiency in parallel applications. One is a lack of enough tasks, so that some slaves are starverd for work while waiting for the next task input. A second reson is that the ratio of communication time to compilation time is too large. The second case, poor communication efficiency, is the more common one.

The communication efficiency can be formally defined as the ratio of the time to execute a task by the time taken for the master to send an initial task message to a slave plus the time for the slave to send back a result message. A good way to diagnose your efficiency is to execute `MasterSlaveStats()` after executing `MasterSlave()`.

1 ▶ `MasterSlaveStats()` F

This function currently returns statistics in the form of a list of two records. The first record provides the global information:

`MStime`
: total runtime (as measured by `Runtime()`),

`MSnrTasks`
: total number of tasks (not including `REDO` or `UPDATE`,

`MSnrUpdates`
: total number of times action `UPDATE` was returned, and

`MSnrRedos`
: total number of times action `REDO` was returned.

The second record provides per-slave information:

`total`
: total time spent on tasks, not including `UpdateSharedData()`,

`num`
: number of initial tasks, `REDO` and `CONTINUATION()` actions,

`ave_ms`
: the value of `QuoInt(1000*total,num)` in GAP, and

`max`
: maximum time spent on a task, in seconds.

Note that for purposes of the per-slave statistics, separate time intervals are recorded for each initial task, `REDO` action, and `CONTINUATION()` action. The time for `UpdateSharedData()` is not included in these statistics. This is because after an `UPDATE` action, the slave does not reply to the master to acknowledge when the update was completed.

Notes:

Poor communication efficiency is typically caused either by too small a task execution time (which would be the case in the example of section or too large a message (in which case the communication time is too long). We first consider execution times that are too small.

On many Ethernet installations, the communication time is about 0.01 seconds to send and receive small messages (less than 1 Kb). Hence the task should be adjusted to consume at least this much CPU time. If the naturally defined task requires less than 0.01 seconds, the user can often group together several consecutive tasks, and send them as a single larger task. For example, in the factorization problem of section, one might modify *DoTask*`()` to test the next 1000 numbers as factors and modify *SubmitTaskInput*`()` to increment `counter` by 1000.

There is another easy trick that often improves communication efficiency. This is to set up more than one slave process on each processor. This improves the communication efficiency because during much of the typical 0.01 seconds of communication time the CPU has off-loaded the job onto a coprocessor. Hence, having a second slave process running its own task on the CPU while a first process is concerned with communication allows one to **overlap communication with computation**.

We next consider the case of messages that are too large. In this case, it is important to structure the problem appropriately. The task architecture is intended to be especially adaptable to this case. The philosophy is to minimize communication time by duplicating much of the execution time on each processor.

After the initial data structure has been built, it will usually be modified as a result of the computation. In order to again minimize communication, the result of a task, which is typically passed to *UpdateSharedData*`()`, should consist of the minimum information needed to update the global data structure. Each process can then perform this update in parallel.

6.3 Checkpointing in TOP-C

Any long-running computation **must** be concerned with checkpointing. The TOP-C model also provides a simple model for checkpointing. The key observation is that the master process always has the latest state of the computation, and the information in the master process is sufficient to reconstruct any ongoing computation. Any application may take advantage of this by checkpointing the necessary information either in the application routine, *SubmitTaskInput*`()` or in *CheckTaskResult*`()`.

A simple way to checkpoint is to record:

- the current data in the TOP-C shared data;
- any private global data residing only in the master process; and
- the inputs to any tasks that are still pending.

This model for checkpointing assumes that your program has no `CONTINUATION()` actions. If you use `CONTINUATION()` actions, then you may require a more complex model for checkpointing.

1 ▶ `MasterSlavePendingTaskInputs()` F

This function returns a GAP list (with holes) of all pending task inputs. If slave i is currently working on a task, index i of the list will record that task. If slave i is currently idle or executing *UpdateSharedData*`()`, then there will be a hole at index i. This function is available for use within either the application routine *SubmitTaskInput*`()`, or *CheckTaskResult*`()`, as specified in the parameters of your call to `MasterSlave()`. (Of course, your application may be using a name other than *SubmitTaskInput*`()` or *CheckTaskResult*`()` in the parameters of `MasterSlave()`.)

6.4 When Should a Slave Process be Considered Dead?

An important question for long-running computations, is when to decide that a slave process is dead. For our purposes, **dead** is not a well-defined concept. If a user on the remote machine decides to re-boot, it is clear that any slave processes residing on that machine should be declared dead. However, suppose there is temporary congestion on the network making the slave unavailable. Suppose that another user on the remote machine has started up many processes consuming many resources, and the TOP-C slave process is being starved for CPU time or for RAM. Perhaps the most difficult case of all to decide is if one particular TOP-C task requires ten times as much time as all other tasks. This last example is conceivable if, for example, each task consists of factoring a different large integer.

Hence, our implementation of the TOP-C model will employ the following **heuristic** in a future version, to decide if a task is dead. You may wish to employ this heuristic now, if you have a demanding application. We use the ParGAP function, `UNIX_Realtime()`, to keep track of how much time has been spent on a task

(based on "wall clock time", and not on CPU time). If a task has taken `slaveTaskTimeFactor` times as much time as the longest task so far, then it becomes a candidate for being declared dead. The GAP variable `slaveTaskTimeFactor` is initially set to the default value of 2.

Once a slave process becomes a candidate for being declared dead, `MasterSlave()` will create a second version of the same task, with the same task input as the original task. `MasterSlave()` will then record which task finishes first. If the original version finishes first, then the second version of the task is ignored, and the slave process executing the original task is no longer considered a candidate for death.

If, however, the second version of the task finishes before the original version, then the time for the second task is recorded. Further, the output from the second task will be used, and any output resulting from the original task will be ignored. `MasterSlave()` then periodically checks until the ration of the time spent so far on the original version of the task is at least `slaveTaskTimeFactor` times greater than the time spent on the second version of the task, then the process executing the original version of the task is then declared dead. No further messages from the process executing original task will be recognized and no further messages will be sent to that slave process.

A future version of this distribution will include direct support for this heuristic. A customized version of it may be used now, by taking advantage of the ParGAP routine, `UNIX_Realtime()`. In addition, a future version of this distribution may include the ability to start new slave processes in an ongoing computation. The reference [CG98] describes how this was done in a C implementation, and why this concept fits naturally with the TOP-C model.

7 MPI commands and UNIX system calls in ParGAP

This chapter can be safely ignored on a first reading, and maybe permanently. It is for application programmers who wish to develop their own low-level message-based parallel application. The additional UNIX system calls in ParGAP may also be useful in some applications.

7.1 Tutorial introduction to the MPI C library

This section lists some of the more common message passing commands, followed by a short MPI example. The next section (7.2) contains more (but by no means all) of the MPI commands and some UNIX system calls. The ParGAP binding provides a simplified form that makes interactive usage easier. This section describes the original MPI binding in C, with some comments about the interactive versions provided in ParGAP. (The MPI standard includes a binding both to C and to FORTRAN.)

Even if your ultimate goal is a standalone C-based application, it is useful to prototype your application with equivalent commands executed interactively within ParGAP. Note that this distribution includes a subdirectory `mpinu`, which provides a subset MPI implementation in C with a small *footprint*. It consists of a C library, `libmpi.a`, and an include file `mpi.h`. The library is approximately 150 KB. The subdirectory can be consulted for further details.

We first briefly explain some MPI concepts.

rank:

The **rank** of an MPI process is a unique ID number associated with the process. By convention, the console (master) process has rank 0. The ranks of the process are guaranteed by MPI to form a consecutive, ascending sequence of integers, starting with 0.

tag:

Each message has associated with it a non-negative integer **tag** specified by the application. Our interface allows you to ignore tags by letting them take on default values. Typical application uses for tags are either to choose consecutive integers in order to guarantee that all messages can be re-assembled in sequence, or to choose a fixed set of constant tags, each constant associated with another type of message. In the latter case, one might have integers for a `QUIT_TAG`, an `INTEGER_ARRAY_TAG`, `START_TASK2_TAG`, etc. In fact, our implementation of the Slave Listener and `MasterSlave()` specifically uses certain tags of value 1000 and higher for these purposes. Hence, application routines that do use tags should restrict themselves to tags `[0..999]`.

communicator:

A **communicator** in MPI serves the purpose of a namespace. Most MPI commands require a communicator argument to specify the namespace. MPI starts up with a default namespace, `MPI_COMM_WORLD`. The ParGAP implementation always assumes that single namespace. A namespace is important in MPI to build modules and library routines, so that a thread may distinguish messages meant for itself, or to catch errors of cross-communication between two modules.

message:
Each message in MPI is typically implemented to include fields for the source rank, destination rank (optional), tag, communicator, count, and an array of data. The *count* field specifies the length of the array. MPI guarantees that messages are non-overtaking, in the sense that if two messages are sent from a single source process to the same destination process, then it is guaranteed that the first process sent will be the first one to arrive, and will be received or probed first from the queue.

other:
MPI also has concepts of *datatype*, *derived datatype*, *group*, *topology*, etc. This implementation defaults those values, so that *datatype* is always a character (hence the use of strings in ParGAP), no *derived datatypes* are implemented, *group* is always consistent with `MPI_COMM_WORLD`, and *topology* is the fully connected topology.

communication:
This implementation implements only point-to-point communication (always blocking receives, except for `MPI_Iprobe`, and sends can be blocking or not, according to the default underlying sockets).

collective communication:
The MPI standard also provides for **collective communication**, which sets up a barrier in which all process within the named communicator must participate. One process is distinguished as the **root** process in cases of asymmetric usage. ParGAP does not implement any collective communication (although you can easily emulate it using a sequence of point-to-point commands). The MPI subset distribution (in ParGAP's `mpinu` directory) does provide some commands for collective communication. Examples of MPI collective communication commands are `MPI_Bcast` (broadcast), `MPI_Gather` (place an entry from each process in an array residing on the root process), `MPI_Scatter` (inverse of gather), `MPI_Reduce` (execute a commutative, associative function with an entry from each process and store on root; example functions are `sum`, `and`, `xor`, etc.

dynamic processes:
The newer MPI-2 standard allows for the dynamic creation of new processes on new processors in an ongoing MPI computation. The standard is silent on whether an MPI session should be aborted if one of its member processes dies, and the MPI standard provides no mechanism to recognize such a dead process. Part of the reason for this silence is that much of the ancestry of MPI lies in dedicated parallel computers for which it would be unusual for one process or processor to die.

Here is a short extract of MPI code to illustrate its flavor. It illustrates the C equivalents of the following ParGAP commands. Note that the ParGAP versions noted here take fewer parameters than their C-based cousins, and ParGAP includes defaults for some optional parameters.

1 ▶ `MPI_Init()` [called for you automatically when ParGAP is loaded] F
▶ `MPI_Finalize()` [called for you automatically when GAP quits] F
▶ `MPI_Comm_rank()` F
▶ `MPI_Get_count()` F
▶ `MPI_Get_source()` F
▶ `MPI_Get_tag()` F
▶ `MPI_Comm_size()` F
▶ `MPI_Send(` *string buf*`,` *int dest*`[,` *int tag = 0* `] )` F
▶ `MPI_Recv(` *string buf* `[,` *int source = MPI_ANY_SOURCE*`[,` *int tag = MPI_ANY_TAG* `] ] )` F
▶ `MPI_Probe( [` *int source = MPI_ANY_SOURCE*`[,` *int tag = MPI_ANY_TAG* `] ] )` F

Many of the above commands have analogues at a higher level in section 2.1 as `GetLastMsgSource()`, `GetLastMsgTag()`, `MPI_Comm_size() = TOPCnumSlaves + 1`, `SendMsg()`, `RecvMsg()` and `ProbeMsg()`.

```
#include <stdlib.h>
#include <mpi.h>
```

```
#define MYCOUNT 5
#define INT_TAG 1

main( int argc, char *argv[] )
{
  int myrank;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &myrank );

  if ( myrank == 0 ) {
    int mysize, dest, i;
    int buf;
    printf("My rank (master):  %d\n", myrank);
    for ( i=0; i<MYCOUNT; i++ )
      buf = 5;
    MPI_Comm_size( MPI_COMM_WORLD, &mysize );
    printf("Size:  %d\n", mysize);
    for ( dest=1; dest< mysize; dest++ )
      MPI_Send( &buf, MYCOUNT, MPI_INT, dest, INT_TAG, MPI_COMM_WORLD );
  } else {
    int i;
    MPI_Status status;
    int source;
    int count;
    int *buf;
    printf("My rank (slave):  %d\n", myrank);

    MPI_Probe( MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status );
    printf( "Message pending with rank %d and tag %d.\n",
            status.MPI_SOURCE, status.MPI_TAG );
    if ( status.MPI_TAG != INT_TAG )
      printf("Error: Bad tag.\n"); exit(1);
    MPI_Get_count( &status, MPI_INT, &count );
    printf( "The count of how many data units (MPI_INT) is:  %d.\n", count );
    buf = (int *)malloc( count * sizeof(int) );

    source = status.MPI_SOURCE;
    MPI_Recv( buf, count, MPI_INT, source, INT_TAG, MPI_COMM_WORLD, &status );
    for ( i=0; i<MYCOUNT; i++ )
      if ( *buf != 5 ) printf("error:  buf[%d] != 5\n", i);
    printf("slave %d done.\n", myrank);
    }
  MPI_Finalize();
  exit(0);
}
```

Even in this simplistic example, it was important to specify

```
MPI_Recv( buf, count, MPI_INT, source, INT_TAG, MPI_COMM_WORLD, &status );
```

and not to use `MPI_ANY_SOURCE` instead of the known source. Although this alternative would often work, there is a danger that there might be a second incoming message from a different source that arrives between the calls to `MPI_Probe()` and `MPI_Recv()`. In such an event, MPI would be free to receive the second message

in `MPI_Recv()`, even though the appropriate count of the second message is likely to be different, thus risking an overflow of the `buf` buffer.

Other typical bugs in MPI programs are:

- Incorrectly matching corresponding sends and receives or having more or fewer sends than receives due to the logic of multiple sends and receives within distinct loops.
- Reaching deadlock because all active processes have blocking calls to `MPI_Recv()` while no process has yet reached code that executes `MPI_Send()`.
- Incorrect use of barriers in collective communication, whereby one process might execute:

```
MPI_Send( buf, count, datatype, dest, tag, COMM_1 );
MPI_Bcast( buffer, count, datatype, root, COMM_2 );
```

  and a second executes

```
MPI_Bcast( buffer, count, datatype, root, COMM_2 );
MPI_Recv( buf, count, datatype, dest, tag, COMM_1, status );
```

  If the call to `MPI_Send()` is blocking (as is the case for long messages in the case of many implementations), then the first process will block at `MPI_Send()` while the second blocks at 'MPI_Bcast()'. This happens even though they use distinct communicators, and the send-receive communication would not normally interact with the broadcast communication.

Much of the TOP-C method in ParGAP (see chapters 3 and 5) was developed precisely to make errors like those above syntactically impossible. The slave listener layer also does some additional work to keep track of the `status` that was last received and other bookkeeping. Additionally, the TOP-C method was designed to provide a higher level, task-oriented "language", which would naturally lead the application programmer into designing an efficient high level algorithm.

7.2 Other low level commands

Here is a complete listing of the low level commands available in ParGAP. Some of these commands were documented elsewhere. The remaining ones are not recommended for most users. Nevertheless, it may be useful to others for more sophisticated applications.

For most of these commands, the source code is the ultimat documentation. However, you may be able to guess at the meaning of many of them based on their names and their similarity UNIX system calls (in the case of `UNIX_...`) or MPI commands (in the case of `MPI...`). Some of the commands will also show you their calling parameters if called with the wrong number of arguments. Many of the MPI commands have simplified calling parameters with certain arguments optional or set to defaults, making them easier for interactive use.

1 ▶ `UNIX_MakeString(` *len* `)` F
▶ `UNIX_DirectoryCurrent()` [Defined in 'pkg/pargap/lib/slavelist.g'] F
▶ `UNIX_Chdir(` *string* `)` F
▶ `UNIX_FflushStdout()` F
▶ `UNIX_Catch(` *function*, *return_val* `)` F
▶ `UNIX_Throw()` F
▶ `UNIX_Getpid()` F
▶ `UNIX_Hostname()` F
▶ `UNIX_Alarm(` *seconds* `)` F
▶ `UNIX_Realtime()` F
▶ `UNIX_Nice(` *priority* `)` F
▶ `UNIX_LimitRss(` *bytes_of_ram* `)` [= setrlimit(RLIMIT_RSS, ...)] F

2 ▶ `MPI_Init()` F
▶ `MPI_Initialized()` F
▶ `MPI_Finalize()` F
▶ `MPI_Comm_rank()` F
▶ `MPI_Get_count()` F
▶ `MPI_Get_source()` F
▶ `MPI_Get_tag()` F
▶ `MPI_Comm_size()` F
▶ `MPI_World_size()` F
▶ `MPI_Error_string(` *errorcode* `)` F
▶ `MPI_Get_processor_name()` F
▶ `MPI_Attr_get(` *keyval* `)` F
▶ `MPI_Abort(` *errorcode* `)` F
▶ `MPI_Send(` *string buf* `,` *int dest*`[,` *int tag = 0* `] )` F
▶ `MPI_Recv(` *string buf* `[,` *int source = MPI_ANY_SOURCE*`[,` *int tag = MPI_ANY_TAG* `] ] )` F
▶ `MPI_Probe( [` *int source = MPI_ANY_SOURCE*`[,` *int tag = MPI_ANY_TAG* `] ] )` F
▶ `MPI_Iprobe( [` *int source = MPI_ANY_SOURCE*`[,` *int tag = MPI_ANY_TAG* `] ] )` F

3 ▶ `MPI_ANY_SOURCE` V
▶ `MPI_ANY_TAG` V
▶ `MPI_COMM_WORLD` V
▶ `MPI_TAG_UB` V
▶ `MPI_HOST` V
▶ `MPI_IO` V

8 Comments?

COMMENTS SOLICITED:

I welcome comments on how well the TOP-C parallel model fits other applications. I am also interested in building up a library of ParGAP programs that can be made available to other users. Finally, one ingredient in making a system usabe is a good choice of names that makes the purpose of various commands obvious to a new user. I welcome suggestions. For example, `ParEval`, `BroadcastMsg`, `SendRecvMsg`, and `SendMsg` all have related functionalities. The same can be said of `RecvMsg`, `ProbeMsg`, `ProbeMsgNonBlocking`. Hence, a more orthogonal naming scheme might be easier.

If you are interested in "looking over my shoulder", you might also want to inspect some of my random scribbling in the ParGAP's `etc` subdirectory.

Bibliography