# Parallel GAP: Mature Interactive Parallel Computing

*Gene Cooperman*[*]

**Abstract.** Parallel GAP (ParGAP) is now a mature system to support parallel applications. No prior experience with parallel programming is needed in order to quickly develop new parallel applications. This article provides a description of some of the features of ParGAP that can greatly ease the challenge of writing a parallel program. As is true with standard GAP, ParGAP can be run in a completely interactive manner. This simple interactive usage is illustrated by a short program that implements "parallel streaming", running distinct algorithms on the same problem, but under different processes. One then returns the first answer to be produced, and interrupts the other processes. This usage of ParGAP can be useful even on a single computer.

1991 Mathematics Subject Classification: primary 68N99, secondary 20-01, 20-04.

## 1. Introduction

The acceptance of ParGAP (Parallel GAP) as a GAP share package marks ParGAP's transition into a mature package to support parallel computation. The goals of ParGAP are to support *cheap* parallel computing suitable, for example, for a network of workstations or Beowulf cluster. It employs a simple programmer's model of parallelism that requires minimal learning time.

An earlier version is reported on, under the name of GAP/MPI [**Co97**]. Since then, the defining concepts of ParGAP have been both simplified and extended. Restrictions in the earlier version have also been considerably reduced. For example, the earlier GAP/MPI has had a tendency to crash or hang following a CTRL-C (interrupt) input by the user.

In an implementation on a network employing 10 Mb/s Ethernet, one often observes a message latency time of 1/100 of a second, and in the author's experience, the latency can sometimes approach 1/10 of a second under realistic

*Supported in part by NSF Grant CCR-9732330.

conditions. In a fast Ethernet (100 Mb/s), one does *not* observe times that are ten times faster, since many of the delays occur in the workstation itself, as a message is copied from one buffer to another and then sent out as multiple packets.

Hence, a key issue for ParGAP is latency tolerance. The ParGAP model of parallelism is the same as that for TOP-C [**Co96**] (Task-Oriented Parallel C). Hence, one can also combine TOP-C with GAP's ability to link a standalone C program, and create a C binary that is callable from GAP and that will execute a parallel program in C. In general, the features of the ParGAP/TOP-C parallel model include:

- checkpointing;
- natural load balancing;
- latency tolerance;
- overlap of computation and communication;
- heterogeneous parallelism (multiple CPU and O/S architectures);
- robustness (when slave processes die, after interrupt, etc.); and
- small implementation (1260 lines of GAP code, and 757 lines of C code).

Experimental versions of this package have at various times included the following additional capabilities. (The citations describe the original implementations of these features in C.)

- dynamic attachment of new processors [**CG98**];
- Meta-computing on the Web (attachment of remote processes) [**CG98**]; and
- an SMP version [**CH97**].

The TOP-C model of parallelism has been used in several computations that were the largest of their kind at the time. These include

- construction of a permutation representation for Lyons's group acting on 9,606,125 points given a representation in GL(111,5) [**CFTY97**];
- coset enumeration of Lyons's group on a 4-processor SMP architecture (8,835,156 cosets) [**CH97**];
- condensation of Thompson's group to dimension 5,693 given a representation in GL(112,2) [**CHLM97**]; and

- construction of a permutation representation for $J_4$ on 173,067,389 points given a representation in GL(1333,11) [**CLMW99**].

The rest of this paper summarizes some of the novel features of ParGAP. For the full details of ParGAP, it is recommended to read the ParGAP manual, which contains approximately 50 pages, including many further examples. The entire ParGAP distribution, including the manual, can be downloaded from

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

The distribution is a compressed file of approximately 256 Kilobytes.

Section 2 covers trivial parallelism (the simplest and most common form of parallel application). This is followed in Section 3 by a description of how to use ParGAP interactively, and Section 4 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 5. Finally, the advantages of ParGAP over a lower level parallel standard, such as MPI (Message Passing Interface) [**GLS94,MPI94**], are briefly reviewed in Section 6.

## 2. 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, which is called `ParList()` in ParGAP.

The following is an example of a ParGAP session that uses the `ParList()` function. Observe the use of `BroadcastMsg()` 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 ),
        rec( order := 6, degree := 3, baseSize := 2 ),
        .
        .
    <GAP output truncated>
```

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 slave) 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()` (to be implemented in the next version of ParGAP, ParGAP-1.1.0) will specify 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 with ParGAP-1.1.0, one will be able to execute the following.

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

## 3. 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 [**Co95**], 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 share packages. After extracting the ParGAP files, one invokes `configure` and `make`. One next writes a file with a list of slave processors to attach, and an indication of where ParGAP is located on that processor. In ParGAP, this file is called `procgroup`, and must reside in the current directory. An example 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/gap4r1/pkg/pargap/bin/pargap.sh
localhost 1 /proj/sparc/gap4r1/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/gap4r1/pkg/pargap/bin/pargap.sh
```

Next, instead of invoking `gap`, one then invokes the supplied shell script, `pargapmpi.sh`, (to be renamed `pargap.sh` in ParGAP-1.1.0). Observe that unlike other GAP packages, there is no need to call `RequirePackage()` function from within a GAP session. 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:

`pargapmpi.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 to slave `i`. Slave `i` then evaluates the 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.

We now illustrate the usage of the commands mentioned above with a sample ParGAP session, for which a procgroup file has defined two slaves.

```
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 ‘\n’
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 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\")" );
```

```
/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 above to get fresh prompt.
gap> RecvMsg( 2 ); # "No Value" msg from last SendMsg() command
"<no_return_val>"
gap> SendMsg( "a", 1 ); # 'a' is known on slave 1
gap> RecvMsg( 1 );
45
gap> myfnc := function() return 42; end;;
gap> # Use PrintToString() to define myfnc on 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 environment read into master and slaves.
gap> ParRead( "/home/gene/myprogram.g");
```

## 4. 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. The GAP 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 $i^{\mathrm{th}}$ slave to evaluate the function on the $i^{\mathrm{th}}$ 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.

```
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". Assume that `ParFirstResult()` is defined on the master process and that we have two slaves.

```
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 multiple polynomial quadratic sieve algorithm (MPQS), and the other uses the continued fraction algorithm (CFRAC). We demonstrate the "streaming" of these algorithms in determining the factorization of $35!+1$. It is important to ensure that the "FactInt" package has been loaded on the slaves. Currently, one must add `RequirePackage("FactInt");` to one's `.gaprc` file before beginning our `ParGAP` session, in order to ensure this. This will be fixed in a future version. For more information, see `?FactInt` within GAP on the FactInt package [**Ko99**].

```
gap> # RequirePackage("FactInt"); in .gaprc to define on slaves.
gap> # Now, define StreamingFactInt() on each of the slaves.
gap> BroadcastMsg( PrintToString( "StreamingFactInt :=",
```

```
>                                StreamingFactInt ) );
gap> ParFirstResult([1,2], i->StreamingFactInt(i, 35) );
... resetting ...
[ "CFRAC ALGORITHM",
  [ [ 137, 379, 17839, 340825649, 32731815563800396289317 ],
    [  ] ] ]
```

## 5. 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. In this paper, we provide only the basics of the parallel model. The parallel model was described in more detail in [**Co97**], and a still more detailed description is contained in the manual.

For non-trivial parallelism, ParGAP uses the TOP-C model [**Co96**]. ParGAP typically invokes the TOP-C model via a command such as

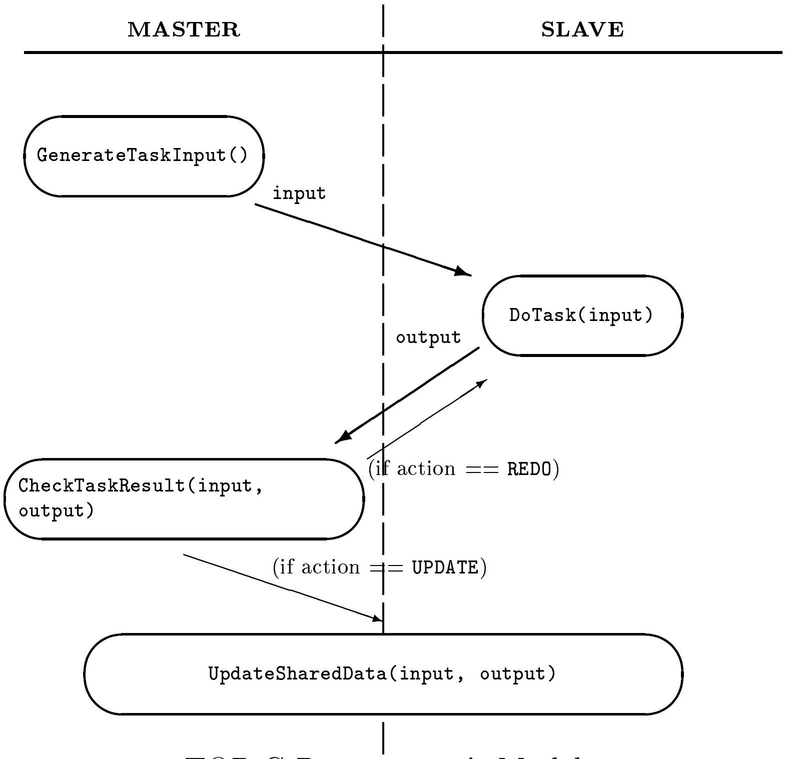
```
MasterSlave( GenerateTaskInput, DoTask, CheckTaskResult,
             UpdateSharedData );
```

The four routines, `GenerateTaskInput`, `DoTask`, `CheckTaskResult`, and `UpdateSharedData` are "callback" functions written by the application programmer, and the names of those callback functions are arbitrary. (In fact, we illustrate using the choice of callback function names from the upcoming ParGAP-1.1 manual. Although the ParGAP-1.0 manual uses slightly different names for its examples, the illustrations here run verbatim in either version.)

The only ParGAP command above is `MasterSlave()`. A task is an arbitrary routine (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 below gives some idea of the flow of control as a task is processed.

The diagram above 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




TOP-C Programmer's Model
(Life Cycle of a Task)

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). (In ParGAP-1.0.x, uses `UPDATE_ENVIRONMENT`. ParGAP-1.1.0 will offer the alternative `UPDATE_SHARED_DATA`. We standardize on the latter in this description.)

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 2.

```
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`. 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 application routine 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 application routine CheckTaskResult(), it is the first argument to that routine. See the ParGAP manual for more sophisticated methods to provide high concurrency of the slave tasks, which go beyond the use of the IsUpToDate() utility.

## 6. Comparison of ParGAP/TOP-C with MPI (Message Passing Interface)

There already exists a standard for parallel programming using messages. It is MPI (Message Passing Interface) [**GLS94,MPI94**]. Hence, it is reasonable to ask how ParGAP compares. ParGAP is in fact written on top of a subset MPI implementation that is included with the distribution. However, MPI was intended as a lower level standard. Hence, the MPI standard defines a binding to FORTRAN and to C, but not to any interactive language, such as GAP, Visual Basic, etc. This is due to the history of MPI, which was originally intended as a standard specification for message passing, that could then be implemented by each computer vendor, who might also provide specialized hardware assistance to improve the latency and throughput of messages.

Some ParGAP issues were not covered by the original MPI standard because they do not occur in compiled languages. Some examples are:

- a slave process executes error and enters break loop;
- a slave process enters infinite loop;
- a user types ˆC (interrupt); and

- a runaway slave process occurs that doesn't recognize `quit;`.

See [**Co95**] for a discussion of these interactive issues in the context of an implementation of the TOP-C model for LISP. At a higher level, the MPI programmer is still required to carry out several error-prone tasks. The issues below assume some knowledge of MPI.

- Each send command must be matched with a corresponding receive command.

- One must avoid deadlock situations in which every process is blocked on an outstanding receive request, and so no process can reach the corresponding send command.

- One must avoid deadlock arising because a receive is executed on one process before a barrier (such as occurs in MPI's collective communication), and a send is executed on a second process after the barrier.

Finally, there are still more subtle MPI traps for the unwary application programmer, such as the example below. We illustrate the trap using C syntax, while writing "`...`" for arguments that do not matter in this example. A bug arises because MPI allows the two invocations of `MPI_ANY_SOURCE` to refer to two distinct source (slave) processes. Hence, the call to `MPI_Probe()` will fill in the status for one process, and a buffer will be allocated based on the pending message from that first process. However the call to `MPI_Recv()` employs `MPI_ANY_SOURCE` a second time, and can cause a different, larger message from a second slave process to be received and truncated to fit in the buffer provided.

```
MPI_Probe(MPI_ANY_SOURCE, ..., &status);
MPI_Get_size( &status, datatype, &count);
buf = malloc(count * 4);
MPI_Recv(buf, count, datatype, MPI_ANY_SOURCE, ...);
```

## 7. Acknowledgement

The author gratefully acknowledges the many valuable suggestions and careful reading of the reviewer. The detailed comments not only improved the paper, but also provided insights into how better to document future versions of the ParGAP distribution. I would also like to acknowledge the encouragement and feedback from many in the community.

## References

[Co97] G. Cooperman, "GAP/MPI: Facilitating Parallelism", *Proc. of DIMACS Workshop on Groups and Computation II* **28**, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, L. Finkelstein and W.M. Kantor (eds.), AMS, Providence, RI, 1997, pp. 69–84.

[Co96] G. Cooperman, "TOP-C: A Task-Oriented Parallel C Interface", $5^{th}$ *International Symposium on High Performance Distributed Computing* (HPDC-5), IEEE Press, 1996, pp. 141–150.

[Co95] G. Cooperman, "STAR/MPI: Binding a Parallel Library to Interactive Symbolic Algebra Systems", *Proc. of International Symposium on Symbolic and Algebraic Computation (ISSAC '95)*, ACM Press, pp. 126–132.

[CFTY97] G. Cooperman, L.Finkelstein, M.Tselman and B.York, Constructing Permutation Representations for Matrix Groups, *J. Symbolic Computation* **24**, 1997, pp. 1–18.

[CG98] G. Cooperman and V. Grinberg, "TOP-WEB: Task-Oriented Metacomputing on the WEB", *International Journal of Parallel and Distributed Systems and Networks* **1**, 1998, pp. 184–192; a shorter version appears as: "TOP-WEB: Task-Oriented Metacomputing on the Web", G. Cooperman and V. Grinberg, *Proceedings of Ninth IASTED International Conference on Parallel and Distributed Computing and Systems* (PDCS-97), IASTED/Acta Press, Anaheim, 1997, pp. 279–286.

[CH97] G. Cooperman and G. Havas, Practical parallel coset enumeration, *Proc. of Workshop on High Performance Computation and Gigabit Local Area Networks*, G. Cooperman, G. Michler and H. Vinck (eds.), Lecture notes in control and information sciences **226**, Springer Verlag, 1997, pp. 15–27.

[CHLM97] G. Cooperman, G. Hiss, K. Lux, and Jürgen Müller, The Brauer tree of the principal 19-block of the sporadic simple Thompson group, *J. of Experimental Mathematics* **6**(4), 1997, pp. 293–300.

[CLMW99] G. Cooperman, W. Lempken, G. Michler and M. Weller, "A New Existence Proof of Janko's Simple Group $J_4$", *Progress In Mathematics* **173**, 1999, pp. 161–175.

[CT2] G. Cooperman and M. Tselman, New Sequential and Parallel Algorithms for Generating High Dimension Hecke Algebras using the Condensation Technique, *Proc. of International Symposium on Symbolic and Algebraic Computation (ISSAC '96)*, ACM Press, 155–160.

[GLS94] W. Gropp, E. Lusk and A. Skjellum, *Using MPI*, MIT Press, 1994.

[Kn91] D.E. Knuth, "Notes on Efficient Representation of Perm Groups" *Combinatorica* **11** (1991), pp. 57–68 (preliminary version circulated since 1981).

[Ko99] S. Kohl, *FactInt*, GAP share package, `http://www-gap.dcs.st-and.ac.uk/~gap/Share/factint.html`.

[MPI94] Message Passing Interface Forum (author), "MPI: A Message-Passing Interface Standard", *International Journal of Supercomputing Applications* **8**, Number 3/4, 1994.

[Sc95] M. Schönert et al., *GAP — Groups, Algorithms and Programming* (Manual), Lehrstuhl D für Mathematik, RWTH, Aachen, Germany, 1995.

[Si71] C.C. Sims, "Computation with Permutation Groups", in *Proc. Second Symposium on Symbolic and Algebraic Manipulation*, edited by S.R. Petrick, ACM Press, New York, 1971, pp. 23–28.

College Of Computer Science
Northeastern University
Boston, MA 02115 / U.S.A.
gene@ccs.neu.edu