DIMACS Series in Discrete Mathematics
and Theoretical Computer Science
Volume **00**, 19xx

GAP/MPI: Facilitating Parallelism

Gene Cooperman

Abstract. The goal of this work is to overcome the learning barriers faced when first using parallelism. Currently, in order to parallelize a system such as GAP, one must embed a message passing library such as MPI, with many routines and many parameters. GAP/MPI provides a simple, task-oriented interface sitting above the MPI library. The system presents the end-user with a single SPMD (single program, multiple data) environment in GAP: an existing, familiar interactive language. In GAP/MPI one describes the end application in terms of high level tasks, which are invoked by a single procedure call in GAP/MPI. This eliminates the complexities of a message passing library, such as encoding a message in a suitable data structure, message synchronization, communication topologies and deadlock avoidance.

1. Introduction

GAP/MPI is a package that allows one to do parallel programming on any distributed memory architecture supported by GAP [**9**] and MPI [**5, 8**]. The software package allows one to use a network of UNIX workstations. It functions also over heterogeneous workstation architectures, and over wide area networks. Such networks are particularly attractive in a university environment, where student laboratories with many workstations on a single Ethernet can be used effectively during off-peak periods.

One begins to parallelize a problem by breaking it down into *tasks*. Task descriptions are generated on the master, and assigned to a slave. The slave executes the task and returns the result to the master. As a result of the value returned by the slave, the master may optionally choose to invoke a user-defined routine that updates some user-defined global data structures, the *environment*. The system arranges to call the routine on the master and all slaves, so as to maintain a common environment. The user should modify this environment only in the context of this update routine, so that all processors have a common environment.

It may happen that several slaves may return results to the master at approximately the same time. If at most one of the slaves has a result requiring an update to the environment, then the master need call the routine updating the environment only based on the one result. If more than one slave result requires updating

1991 *Mathematics Subject Classification.* Primary 68N99, Secondary 20-01, 20-04.
The author was partially supported by NSF Grant CCR-9509783.

the environment, then there is a *collision.* In the case of a collision, the master can either sequentialize the tasks (accept one result for update, while asking the other slaves to re-do their tasks in light of the updated environment), or else the master may invoke an application-specific mechanism for combining the multiple results.

A set of tasks are called *weakly inter-dependent*, if there is some number less than one, such that the fraction of tasks experiencing collisions is less than that number. As more slaves are added, if the fraction of tasks experiencing collisions remains bounded above by a number less than one, then it is easy to show that the parallel program experiences asymptotically linear speedup.

The system is based on MPI (Message Passing Interface) [**5, 8**] and GAP (a general purpose system for "Groups, Algorithms and Programming") [**9**]. MPI is a standard for implementation of parallelism via a message passing architecture. It has bindings to both C and FORTRAN. There are several implementations of MPI. The package described here has been implemented using the MPICH implementation [**4**]. Since the MPICH implementation is given as a C library, it was necessary to extend the GAP kernel to include an interface to MPI in the GAP language, along with some GAP code that implements the end-user interface. Since GAP/MPI is implemented primarily using the point-to-point layer of MPI, it should be easy to port GAP/MPI to a different message-passing library.

A general description of GAP/MPI is given first (section 2). The concepts are illustrated using coset enumeration, Gröbner bases and strong generating sets for group membership in permutation groups, in the hope that the reader is familiar with one or more of these cases. The concepts in GAP/MPI are language-independent, and variations of it are described for LISP [**1**] (LISP/MPI) and for C [**2, 3**] (TOP-C). The term STAR/MPI has been used to include LISP/MPI, GAP/MPI and other implementations that bind MPI to an interpreted language using the master-slave architecture described here. Strong generating sets have been implemented in GAP/MPI, while Gröbner bases were implemented in LISP/MPI and coset enumeration was implemented in TOP-C.

The later sections then provide a running example based on a naive integer factorization using the sieve of Eratosthenes. The familiarity and simplicity of this algorithm ensure that the algorithmic details do not obscure the issues of parallelization in the master-slave architecture.

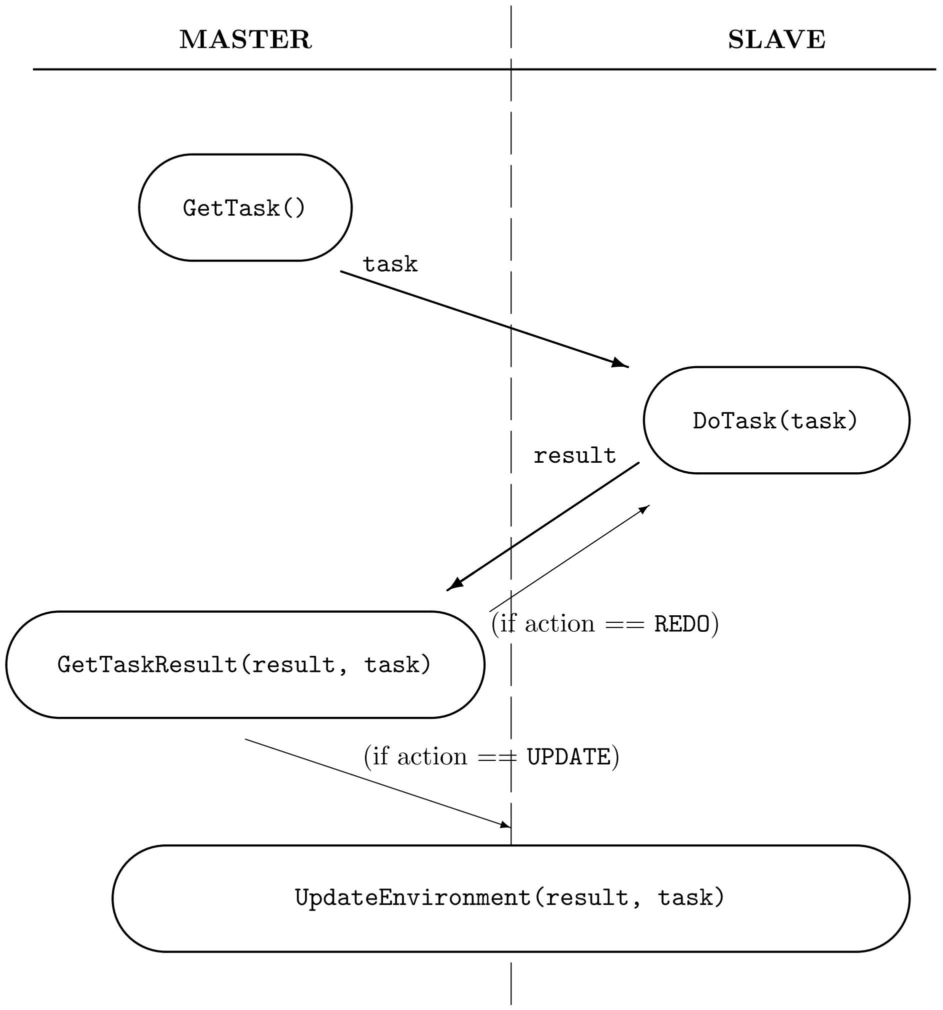
This is followed by some detailed examples in sections 3 and 4 and a general template for programming with GAP/MPI in section 5. Section 6 describes some advanced features, including an alternative interface to the master-slave architecture. That alternative is often attractive when one is parallelizing code consisting of many nested loops. Efficiency considerations are discussed in section 7 and the paper is completed by section 8 on debugging techniques. That section is of independent interest for its simulator of GAP/MPI, that can be run in a standard, sequential GAP.

Debugging parallel programs in a MIMD (multiple instruction, multiple data) architecture has always been a stumbling block in making the transition from sequential to parallel programs. This stumbling block is substantially eased in a master-slave architecture because all messages must go through a single master. So, the master acts to sequentialize all communication, and this can be observed by user.

The software is freely distributed by the author, and will also be available as a share library in GAP. The motivated reader is encouraged to obtain the software package and to try some of the examples from this paper. GAP/MPI is part of a family of languages, all following the same parallel design principles laid out here. Currently, there also exist implementations in LISP [**1**] and C [**2**]. GAP is close enough to a general pseudo-code that the interested reader should have no trouble constructing the same examples in these other languages.

2. Basic Concepts

A *master-slave architecture* consists of a unique processor, designated the *master*, and arbitrarily many other processors, designated *slaves*. Communication is constrained to pass only between master and slaves, and not among slaves. As we shall see, communication concerning a particular task is further constrained, in that the first communication concerning a new task must be initiated by the master. Hence, the master generates new tasks to be done, and the slave executes them. Graphically, one has the following picture.



The primary interface to GAP/MPI is an invocation of the form:

```
MasterSlave( GetTask, DoTask, GetTaskResult, UpdateEnvironment )
```

where the four variables, `GetTask`, `DoTask`, `GetTaskResult` and `UpdateEnvironment`, are bound to functions defined by the user. This invocation causes a parallel program to execute. The life cycle of a single task has been displayed graphically above. In the special case of a single slave, this is equivalent to the sequential pseudo-code, below.

```
while ( NOTASK <> (task = GetTask()) ) do                [on master]
  redo:
  result := DoTask( task );                               [on slave]
  action := GetTaskResult( result, task );               [on master]
  switch ( action )
    case NO_ACTION: /* do nothing */;
    case UPDATE: UpdateEnvironment(result, task);[on master & slave]
    case REDO:   goto redo;
    case CONTINUATION:  ;             [action defined in section 6]
```

The values of the variables `task` and `result` can be arbitrary, and are defined only by the return values of the user functions `GetTask()` and `DoTask()`. In elementary applications of `MasterSlave()`, the action will always be `NO_ACTION`. The reader may wish to consider only this special case on a first reading (in which case the function `UpdateEnvironment()` is not used).

Of the four user-defined functions, the routine `GetTask()` executes on the master, the routine `DoTask()` on a slave, the routine `GetTaskResult()` on the master, and the routine `UpdateEnvironment()` on master and each slave. The system arranges to execute multiple tasks (one on each slave) at the same time.

In addition to the four user-defined functions, the fundamental concepts of GAP/MPI are the task, the result, the action, and the environment.

A *task description* is a user-defined data structure, that is the input to a routine, `DoTask()`. The routine, `DoTask()`, is called with a single argument, the current task description. The routine may also read values from the environment. If a program in GAP/MPI is to be efficient, then the large majority of the CPU time of the algorithm should be spent in calls to `DoTask()`. This implies, in particular, that most calls to `DoTask()` should not require significant updates to the environment.

The return value of `DoTask()` is the *result.* This and the original task description are the input parameters for `GetTaskResult()`. This latter routine returns an *action* that controls further options for processing of the task as depicted in the figure. These options are described in further detail in the later sections.

The *environment* is a set of variables, along with their values, shared among all processes. The set of such variables is chosen by the user according to the following operational definition. The environment is initialized before the first call to `MasterSlave()`, and after that it may be modified only by the routine, `UpdateEnvironment()`. The functions `GetTask()`, `DoTask()` and `GetTaskResult()` may read values from the environment, but only the routine `UpdateEnvironment()` should be allowed to set values in the environment.

In addition to global variables, the environment may include variables that are in the lexical scope of each of the routines `GetTask()`, `DoTask()`, `GetTaskResult()` and `UpdateEnvironment()`. As we shall see in our factorization example, this feature can make the code more modular by making the environment local to the fifth function.

2.1. Examples of Concepts. In the case of implementations of coset enumeration, the environment for coset enumeration might be the coset table (and possibly the subgroup table and relator table). A task description would consist of a coset and a generator. The routine, `DoTask()`, would then process the deduction determined by the coset and generator. The consequences of the deduction, if any, would be returned by `DoTask()`, so that appropriate tables could later be updated. As necessary, one would call `UpdateEnvironment()` to process a resulting coincidence or makes a new deduction.

In the case of construction of Gröbner bases, an environment might be the intermediate base. A task description might consist of a new S-polynomial to test. The work of `DoTask()` would be to reduce the S-polynomial. If the residue were non-trivial, `DoTask()` would return the reduced S-polynomial without modifying the intermediate Gröbner base. The reduced S-polynomial would then be added to the intermediate Gröbner base by `UpdateEnvironment()`.

In the case of construction of strong generating sets for group membership for permutation groups, the environment might be the current intermediate strong generating set along with a set of fundamental orbits and the corresponding Schreier trees. A task description might consist of a Schreier generator. The work of `DoTask()` would be to sift (strip) the Schreier generator through the Schreier trees and return the residue. If the residue were non-trivial, it would be added to the strong generating set by `UpdateEnvironment()`.

In each of the algorithms above, many task instances don't require a call to `UpdateEnvironment()`. Hence, depending on the input, many instances of these algorithms satisfy weak task inter-dependence. It should be intuitively clear that there is a large family of such algorithms that satisfying weak task inter-dependence. Determination of the AG generating sequence of an AG-group (determination of an AG-System) for a power commutator presentation is still another example. Indeed many of these algorithms can be viewed in the general framework of Knuth-Bendix techniques. While the Knuth-Bendix framework provides a rich family of examples for this software, there are still other examples outside of that family, as will be seen in the following examples.

3. A Simple Example

When GAP/MPI begins executing, the user is presented with a "master" GAP process and several slave processes. The number of slave processes and their binding to particular processors is specified in a manner dependent on the particular MPI implementation.

In our package, we base this on MPICH's `procgroup` file. A typical file appears below. This file sets up two slaves on a single processor, highpoint, and one slave on k2. Because GAP/MPI is designed only for distributed memory applications, the second parameter for each slave entry is always 1. MPICH uses numbers other than 1 for more specialized parallel machines, including shared memory machines.

```
# Modify this for the machines at your site
local 0
highpoint.ccs.neu.edu 1 /proj/gal2/gap-mpi/gapmpi
highpoint.ccs.neu.edu 1 /proj/gal2/gap-mpi/gapmpi
```

```
k2.ccs.neu.edu 1 /proj/gal2/gap-mpi/gapmpi
```

By default, user commands are executed only on the master. The `ParEval` command takes any string and executes it as a user command on the master and all slaves, and returns the value returned by the invocation of the master. The string argument is required to be terminated by `";\n"`.

```
ParEval("x:=3;\n");
```

One typically defines the desired functions and initial data structures in a file visible to all processes (e.g.: on workstations using NFS). It is then loaded into the master and all slaves using the `ParEval` command. Using `Read()` alone (without `ParEval()`) would load the file into the master only.

```
ParEval("Read(\"myfile.g\");\n");
```

If there are n processes, one will see GAP's confirmation of the load printed n times.

Within the user file, `"myfile.g"`, one will have defined the routine to be executed in parallel. This routine calls `MasterSlave()`, and `MasterSlave()` then invokes three user-written functions, `GetTask()`, `DoTask()`, and `GetTaskResult()`.

Let us now consider `ParList`, a parallel version of the two-argument form of the GAP function, `List`. It takes arguments `list` and `Fnc` and applies `Fnc` to each element of `list` and returns a list of the results. A natural decomposition leads to defining a task as the application of the `Fnc` argument to a single element of the `list` argument. The result should then be saved and a list of all the results returned. The environment, here, is simply `Fnc`.

```
# ParList defined in "myfile.g" and loaded as described above.
ParList := function ( list, Fnc )
     local counter, result_list, GetTask, DoTask, GetTaskResult;
     counter := 0;
     result_list := [];
     GetTask := function ()
                counter := counter + 1;
                if counter > Length(list) then return NOTASK; fi;
                return counter; end;
     DoTask := function (task)   # task is last value of counter
               return Fnc( list[task] ); end;
     GetTaskResult := function ( result, task )
                      result_list[task] := result;
                      return NO_ACTION; end;
     MasterSlave( GetTask, DoTask, GetTaskResult );
     return result_list;
end;
ParEval( "ParList( [1..100], x->x^2 );\n" ); # returns [1,4,...,10000]
```

Note that we have taken advantage of GAP's ability to define a function within another function. We did this so that the local variable `counter` can be shared by the functions `ParList()` and `GetTask()`, the local variable `result` can be shared by the functions `ParList()` and `GetTaskResult()`, and the arguments to `ParList()`, `list` and `Fnc`, are shared with `DoTask()` on each slave. If one prefers to define all functions globally, then one would have to rewrite the above code to make all variables global, and to introduce new global variables to share values of `list` and `Fnc` between the functions `ParList()` and `GetTaskResult()`.

4. An Extended Example

The typical development of a parallel program is illustrated by an extended example for parallel integer factorization, using the sieve of Eratosthenes. For clarity of exposition, we ignore many possible optimizations, such as stopping the sieve after testing the square root, and we especially ignore the existence of more sophisticated factorization algorithms. The reader who has absorbed the lessons of this section will have no trouble applying them in a more sophisticated manner.

4.1. A naive, parallel primality test. The following code employs a variation of the sieve of Eratosthenes. The environment is the variable, `num_to_factor`. The routine `UpdateEnvironment()` is introduced here and used as described in section 2.

```
MasSlaveIsPrime := function ( num_to_factor )
     local GetTask, DoTask, GetTaskResult, UpdateEnvironment, last_num;
     last_num := 1;
     GetTask := function ()
                last_num := last_num + 1;
                if last_num > num_to_factor then return NOTASK; fi;
                return last_num; end;
     DoTask := function ( num ) # Test if num (= last_num) is a factor
               return num_to_factor mod num = 0; end;
     GetTaskResult := function ( result, num )
                      if result = true then return UPDATE;
                      else return NO_ACTION; fi; end;
     UpdateEnvironment := function( result, num )
                          num_to_factor := 1; end;
     MasterSlave( GetTask, DoTask, GetTaskResult, UpdateEnvironment );
     return num_to_factor <> 1; # true means num_to_factor is prime
end;
ParFactor := function( num_to_factor )
             return ParCall( "MasSlaveIsPrime", num_to_factor ); end;
```

As before, `MasSlaveIsPrime` would be defined in a file such as `"myfile.g"`. The function `ParFactor()` could also be defined there, or defined only on the master. Now, the user need only type `ParFactor(num_to_factor)` instead of `ParEval(...)`. To accomplish this, we invoked a new utility, `ParCall()`. The function `ParCall()` first distributes its arguments to master and all slaves, it then applies the first argument (a string giving the name of a function) to the remaining arguments in each process, and it finally returns the value returned by the invocation on the master.

4.2. Parallel integer factorization. This section generalizes the routine `MasSlaveIsPrime` of the previous section to now tackle factorization, while also introducing several additional features of GAP/MPI. Finding all the factors requires updating an environment in a non-trivial manner. Otherwise, two slaves would separately note that 2 and 4 both divide 12, and there is a danger of storing both factors. Hence, the environment now includes both `num_to_factor` and a list, `factors`.

The code below does not guarantee a prime factorization, but only a factorization into possibly composite numbers. This "bug" arises because one slave may receive its task after another slave, but still return first. The analysis of this bug and its fix are deferred to the following section, in order to emphasize this pitfall of parallel programming.

```
MasSlaveFactors := function ( num_to_factor )
     local GetTask, DoTask, GetTaskResult, UpdateEnvironment,
           last_num, factors;
     factors := [];
     last_num := 1;
     GetTask := function ()
                last_num := last_num + 1;
                if last_num > num_to_factor then return NOTASK; fi;
                return last_num; end;
     DoTask := function ( num ) # Test if num (= last_num) is a factor.
               return num_to_factor mod num = 0; end;
     GetTaskResult := function ( result, num )
                    if result = false then return NO_ACTION; fi;
                    # reset task counter, to re-evaluate later tasks
                    last_num := Minimum( num, last_num );  # (*)
                    return UPDATE; end;  # update num_to_factor
     UpdateEnvironment := function ( result, num )
                          while ( num_to_factor mod num = 0 ) do
                            factors[ Length(factors) + 1 ] := num;
                            num_to_factor := num_to_factor / num;
                          od; end;
     MasterSlave( GetTask, DoTask, GetTaskResult, UpdateEnvironment );
     return factors; end;
ParFactor := function( num_to_factor )
             return ParCall( "MasSlaveFactors", num_to_factor ); end;
```

Just before calling `UpdateEnvironment()`, we reset the counter, `last_num`, back to the task just after the one generating the new factor. In the example, `ParFactor(12)`, this insures that upon discovering 2 as a factor, we will not report both 2 and 4 as factors. Instead, we reset `num_to_factor` to 6 and begin testing factors after 2 again.

Note that the code above causes excessive recomputation. In the computation of `ParFactor(12)`, when a slave discovers that 3 is a factor, then new tasks are generated for testing all factors above 3. But, there are other slaves waiting to report that 5, 7 and 8 are not factors. The current code throws away this information, only to re-compute it later.

To fix this, the routine for `GetTaskResult()` is modified to:

```
GetTaskResult := function ( result, num )
                 if result = false then return NO_ACTION; fi;
                 if not IsUpToDate() then return REDO; fi; # (*)
                 return UPDATE; end;  # update num_to_factor
```

The combination of `IsUpToDate()` and the `REDO` action above is a standard idiom in programming `MasterSlave()`. Roughly, `IsUpToDate()` tests whether `UpdateEnvironment()` has been called since the original call to `GetTask()` that resulted in the current invocation of `GetTaskResult()` (see below for more detail). If there was an intervening call to `UpdateEnvironment()`, then `IsUpToDate()` returns false, and the `REDO` action causes the original task (represented by the variable `num` here) to be re-sent to the original slave process. That slave will then re-compute the result, based on the newly modified value of the environment.

The function `IsUpToDate()` may be called by the end user from within his routine, `GetTaskResult(result, task)`. The value of the parameter `task` was originally generated by a call to `GetTask()`, which had resulted in a call to `DoTask(task)` on a slave, and finally the current call to `GetTaskResult(result, task)` (recall the life cycle of a task in section 2. The function `IsUpToDate()` returns false if and only if `UpdateEnvironment()` has been called since the original call to `GetTask()` corresponding to the current, or most recent, invocation of `GetTaskResult()`.

4.3. A Subtle bug. There is still one bug in the above code. One of the factors returned might be a composite number and not a prime. If `ParFactor(12)` is invoked, causing three slaves to examine in parallel the three factors, 2, 3, and 4, then the slave examining the factor 4 might return first. In this case, `ParFactor(12)` would return `[ 4, 3 ]`. The following modifications fix this bug.

```
# Change line of GetTaskResult commented by "# (*)":
   if not IsUpToDate() and num > Maximum(factors) then return REDO; fi;

# Add to beginning of UpdateEnvironment():
   if num < Maximum(factors) then RemoveMultiples( num );
```

An efficient version would, of course, add at the end of `UpdateEnvironment()` code to save the current value of `Maximum(factors)` in a variable of the enclosing routine `MasSlaveFactors()`. The function `RemoveMultiples(num)` is defined to remove all elements, x, of the array `factors` that are multiples of `num`. It also updates `num_to_factor` by multiplying it by each x. Thus, composite factors are eventually caught and corrected.

Note an interesting phenomenon. Consider again the example, `ParFactor(12)`. In the modified code, if the slave examining the factor 2 were slow to return, then `num_to_factor` might take on values, 12, 4, 1, 2. For this reason, the system will always call `GetTask()` one last time after all slaves have returned, to determine if the last slave has altered the interim determination that the job is done.

5. A General Template for GAP/MPI

With the experience of the previous section, we now attempt to write a general template, that parallelizes many algebraic algorithms, while requiring only that one fill in code fragments from pre-existing sequential code. While a general template seems ambitious, it is a useful exercise for the reader to apply it to his/her own favorite algorithms. This template has already been used to parallelize coset enumeration using the Felsch strategy [**3**] (jointly with G. Havas), the Sims group membership algorithm for permutation groups (jointly with A. Hulpke), and Gröbner bases based on the sequential implementation of G. Zacharias [**11**]. The names of functions and variables in the code below have hopefully been chosen to be suggestive of a wide variety of such algorithms.

```
DataStruct := 0; # declare global variable
MasSlaveComplete := function (dataStruct)
  local nextCritPair, GetNextCritPair, SiftCritPair,
        GetResidue, UpdateDataStruct;
  nextCritPairPtr = NULL;
  InitDataStruct(DataStruct);  # put input in canonical form
  GetNextCritPair := function() ...; return nextCritPairPtr; end;
  SiftCritPair := function(critPair)
                 return Sift(critPair, dataStruct); end;
```

```
    GetResidue := function ( result, task )
                  if IsTrivial( result ) then return NO_ACTION; fi;
                  if not IsUpToDate() then return REDO; fi;
                  return UPDATE; end;
    UpdateDataStruct := function ( result, task ) ...; end;
    MasterSlave( GetNextCritPair, ClashPair, GetResidue, UpdateDataStruct );
    return DataStruct;
  end;
  ParComplete := function( dataStruct )
    return ParCall( "MasSlaveComplete", dataStruct ); end;
```

Of necessity, this code leaves out several application-dependent portions. Nevertheless, it is surprising how much can be written without knowing the specific application. Further, the portions left out can usually be copied from a good sequential implementation of the same application.

The function `InitDataStruct(dataStruct)` assumes that `dataStruct` is initially a data structure representing the algorithmic input, such as generators or presentations of a group, ideal, etc. The function modifies `DataStruct` into a convenient data structure for further processing and possibly defines auxiliary lookup tables needed for efficiency.

The variable `nextCritPairPtr` keeps track of the current critical pair being worked on. That variable is updated by `GetNextCritPair()`, which returns either a new critical pair or the constant `NOTASK`. The new critical pair is tested by the function `Sift(critPair, dataStruct)`, which returns a residue. The function `IsTrivial()` tests if the residue is trivial. If it is not trivial, the residue and the original `critPair` become arguments to `UpdateDataStruct()`, which updates `dataStruct` to reflect the new information.

A key point of this code is that for efficiency reasons the most frequent action returned by this code should be `NO_ACTION`. This corresponds to the idea that the residue should usually be trivial, and the slaves should act as a pre-processing filter to eliminate the trivial critical pairs. Thus, updating the non-trivial critical pairs is still a sequential activity, but the time for such updates should be dominated by the time for sifting critical pairs.

6. Advanced Features

There are several features of GAP/MPI that have been included to handle the full range of applications consistent with the target applications described in section 5. First, one may wish that in the case of an `UPDATE` action, a particularly time- or space-intensive computation should be done on the master only. The routine `IsMaster()` is provided for just these purposes. Typically, it is called inside `UpdateEnvironment()`. It returns "true" for the invocation on the master, and "false" for each invocation on a slave.

Next, `GetTaskResult()` can choose to return a fourth action that is, in a sense, a parametrized variation of the `REDO` action. This is the "continuation". The user returns `continuation(task)`, where task is any user-defined object. This also causes `DoTask(task)` to be called a second time on the same slave as before, but this time with a new object, `task`.

The "continuation" action is a safety hatch for cases where it is difficult to maintain the same environment on both master and slave. This may be the case, for example, if a global variable has a value that requires a great deal of space. One may hesitate to use so much space on each slave either because the slaves do not have as much memory as the master, or because the communication overhead in updating such a large data structure would be prohibitive. In this case, the slave routine, `DoTask()`, can request from the master information that is not kept on the slave by sending to the master a “result” that is actually a request for more information. The master routine, `GetTaskResult()`, should recognize this request and return the desired information as the argument in a “continuation” action.

Next, there are cases when the “natural” algorithm requires frequent recursive calls. This is the case with construction of strong generating sets for permutation groups [**6, 10**] and with construction of an AG generating sequence for AG groups [**7**] (polycyclically presented groups). With care, one can also include a recursive call to `MasterSlave()` inside `GetTaskResult()`.

Finally, there are cases when one is parallelizing existing sequential code with many levels of nested loops. In such circumstances, re-writing the code to extract a proper `GetTask()` function can become a nightmare. For such instances, one has the alternative interface:

```
RawMasterSlave(ParallelLoop, DoTask, GetTaskResult, UpdateEnvironment)
```

Instead of `GetTask()`, one now has a routine `ParallelLoop()`. The routine `ParallelLoop()` has no arguments and returns no value. Instead it should encompass enough of the nested loops so as to compute the next task description. At each point inside the nested loops of the sequential code where one has generated another task description, one should replace the user code to execute the task by a call to the system-defined function `SetTask(task)` with `task` being a data structure corresponding to the new task description. The user-defined routine `DoTask(task)` then contains the code to execute the task. Consider the following brief example for parallel matrix multiplication.

```
# code fragment for sequential matrix multiplication
for i in Length[mat_a[1]] do
  for j in Length[mat_b] do
    sum := 0;
    for k in Length[mat_a] do
      sum := sum + mat_a[i][k] * mat_b[k][j]; do;

# parallel matrix multiplication
ParMatMult( mat_a, mat_b )
  local mat_c;
  mat_c := NullMat( Length[mat_a[1]], Length[mat_b];

  ParallelLoop := function()
    local i, j;
    for i in Length[mat_a[1]] do
      for j in Length[mat_b] do
        SetTask([i,j]); od; od; end;
  DoTask := function( task )
    local i, j, k, sum;
    i := task[1], j := task[2], sum := 0;
    for k in Lenth[mat_a] do
      sum := sum + mat_a[i][k] * mat_b[k][j]; do;
    return sum; end;
  GetTaskResult := function( result, task )
    local i, j;
```

```
      i := task[1], j := task[2];
      mat_c[i][j] := result; end;

    RawMasterSlave(ParallelLoop, DoTask, GetTaskResult,
                   UpdateEnvironment);
    return mat_c;
  end;
```

Internally, even the routine `MasterSlave()` has been implemented in terms of `RawMasterSlave()`. The routine `RawMasterSlave()` makes it easier to parallelize existing code that is implemented in many nested loops, but it has the corresponding drawback of a less elegant user interface.

GAP/MPI is implemented on top of a slave-listener layer. As an alternative to the master-slave architecture, one can use GAPMPI directly at this level. This level supports the commands, `SendCommand()` and `GetResult()`, which are briefly mentioned in section 8, but are not otherwise discussed in this paper.

7. Efficiency Considerations

One of the most common causes of inefficiency in parallel programs is high communication overhead. The communication efficiency can be 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.

Poor communication efficiency is typically caused either by too small a task execution time (which would be the case in the example of section 4) 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 3, one might modify `DoTask()` to test the next 1000 numbers as factors and modify `GetTask()` 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.

Hence, rather than build a large initial global data structure on the master and then send it to the slaves, it is often better to build the initial data structure on each processor independently, since this requires no communication. This is why `InitDataStructure()` from section 5 was executed on the master and all slaves. This requires less communication overhead than executing `InitDataStruct()` once on the master and sending the result to all slaves.

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 `UpdateEnvironment()`, should consist of the minimum information needed to update the global data structure. Each process can then perform this update in parallel.

8. Tracing and Debugging

Code development should initially be done on one master and one slave, thus ensuring that one debugs essentially sequential code. If possible, the master and slave should be the same CPU, so as to minimize network delays and ill effects on other users. When that code works correctly, it can then be tested on two slaves, and finally on all possible slaves.

When an error occurs, perhaps the first resort should be to inspect the values of global variables on the slave. This can be aided by having each slave save appropriate trace information in a global variable. In general, one can execute an arbitrary command. As a particularly simple example, consider interrogating slave number 2 for the usm of its variables `x` and `y`.

```
SendCommand("x+y;\n", 2);
GetResult();
```

The second command returns the result of the computation that was done on the first slave. The pair of commands act as a version of `ParEval()` that operates on a single slave, only. One can optionally omit the second argument, which defaults to 1 (slave number 1).

Internally, these commands are part of a lower layer, the slave-listener layer. This layer may be of independent interest for programming a special-purpose alternative to the master-slave architecture using GAP.

Another easy testing strategy is to trace all messages to and from the master. One can cause each task description to be printed in the order that it is seen by the master by setting the following variable:

```
MasterSlaveTrace := true;
```

If this produces too much output, or not the right kind of information, one can add print statements anywhere, whether on master or slave. The MPICH implementation of MPI arranges for output from all processes to be printed on the user console. When printing from a slave, one should be aware that there may be a delay before a printout from a slave appears on the user console, since it must go through the master processor to reach the user console.

If the 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 the following generalization of the sequential code in section 2.

```
contTask := NULL;
NO_ACTION := 0; UPDATE := 1; REDO := 2; CONTINUATION := 3;
NOTASK := "no_task";  # Should not conflict with user values.
Continuation := function(task)
                contTask := task; return CONTINUATION; end;
IsUpToDate := function() return true; end;

MasterSlave := function(arg)
  local GetTask, DoTask, GetTaskResult, UpdateEnvironment,
        task, result, action;
```

```
GetTask := arg[1]; DoTask := arg[2]; GetTaskResult := arg[3];
if Length(arg) = 4 then UpdateEnvironment := arg[4];
else UpdateEnvironment := NULL; fi;
task := GetTask();
while ( task <> NOTASK ) do
  contTask := NULL;
  result := DoTask(task);
  action := GetTaskResult(result, task);
  if action = NO_ACTION then task := GetTask(); # do nothing
  elif action = UPDATE then
      if UpdateEnvironment <> NULL then
        UpdateEnvironment(result, task); fi;
      task := GetTask();
  elif action = REDO then
    Print("REDO isn't useful in sequential case\n"); return;
  elif action = CONTINUATION then task := contTask;
  else Print("illegal return value of GetTaskResult\n"); return;
  fi;
od; end;
```

In this context, one is debugging a completely sequential program.

9. Conclusion

The emphasis of this work is on ease-of-use, even when this conflicts with the design philosophy of a more general tool for parallelization. Nevertheless, the methodology proposed here serves surprisingly well in parallelizing a large variety of algorithms. It appears to fit especially well with many algebraic algorithms that require the flexibility of a MIMD (Multiple Instruction, Multiple Data) parallel architecture for task parallelism. Section 5, a General Parallel Template, is a useful starting point in a project to parallelize such a sequential algorithm, while section 8 contains a simple sequential emulator for initial testing.

Several further extensions would be of great interest. One would be the use of GAP/MPI. Unfortunately GAP, like most programs with built-in garbage collection, does not support a shared memory architecture, but this would be an attractive generalization. Another useful extension would be the elimination of `ParEval()`. One should instead allow the user to choose to "attach" the user console to the master (default), or to the master and all slaves (ParEval-mode), or to a single slave (for debugging). Such experiments should wait for the upcoming GAP version 4.

10. Acknowledgements

The author thanks Frank Celler, Joachim Neubüser and Martin Schönert for their kind discussions and assistance in understanding the internals of GAP and integrating STAR/MPI with GAP. He also thanks Steven Linton and Eamonn O'Brien for their feedback during testing. The author acknowledges a special debt to Alexander Hulpke, who, in addition to testing GAP/MPI on his own code, had the patience to read through and suggest many variations of the GAP/MPI implementation, documentation, etc. Finally, the author gratefully acknowledges the many detailed and valuable comments of the reviewer.

References

[1] 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.

[2] G. Cooperman, "TOP-C: A Task-Oriented Parallel C Interface", 5^{th} *International Symposium on High Performance Distributed Computing* (HPDC-5), IEEE Press, to appear.

[3] G. Cooperman and G. Havas, "Practical parallel coset enumeration", preprint.

[4] N. Doss, W. Gropp, R. Lusk and A. Skjellum, software at info.mcs.anl.gov in /pub/mpi/mpich-Jul22.tar.gZ, anonymous ftp.

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

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

[7] R. Laue and J. Neubüser and U. Schoenwaelder, Algorithms for Finite Soluble Groups and the SOGOS System, *Computational Group Theory*, Academic Press, 1984, pp. 105–135.

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

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

[10] 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.

[11] G. Zacharias, A Groebner basis implementation in LISP, version 202.

College of Computer Science, Northeastern University, Boston, MA 02115
E-mail address: `gene@ccs.neu.edu`