STAR/MPI: Binding a Parallel Library to Interactive Symbolic Algebra Systems

Gene Cooperman[1]
gene@ccs.neu.edu
College of Computer Science
Northeastern University
Boston, MA 02115 / U.S.A.

Abstract

Many users of symbolic algebra systems have felt the need for greater CPU power. Yet few of them have ventured into parallel programming due to the steep learning curve and the unfamiliar programming environment entailed by such an effort. In an attempt to remedy that situation, the parallel library MPI has been integrated into both GCL (GNU Common LISP) and GAP [14] (a general purpose language for mathematical group theory). These implementations are examples that extend bindings of MPI to interactive languages. (MPI already has bindings to the compiled languages C and FORTRAN.) Further, this binding to an interactive language retains the interactive environment during execution. Further, STAR/MPI represents a blueprint for binding MPI to other interactive languages besides GCL and GAP, from which comes the name STAR/MPI, or $*$/MPI. STAR/MPI includes a simple SPMD architecture on top of this MPI binding. An important class of sequential algorithms is described that can be parallelized with little effort using STAR/MPI architecture. Since GAP is representative of systems for discrete mathematics and LISP is the basis for several symbolic algebra systems with strengths in non-discrete mathematics, it is hoped to gain broad feedback on the issues involved. Although vendor-specific, interactive, parallel languages exist, this appears to be the first attempt at defining a binding of a vendor-independent, portable, parallel library to arbitrary interactive languages.

1 Introduction

The goal of this work is to make parallel programming more accessible to users of symbolic algebra systems and to users of interactive languages in general. This has been done by integrating MPI (Message Passing Interface), a portable, parallel message-passing library, with two interactive languages: GCL (GNU Common LISP) and GAP. The GAP system includes a general purpose language for mathematical group theory, and LISP is the basis for several general-purpose symbolic algebra systems. In addition, a simple master-slave abstraction has been written, so that end-users need not learn any of the details of the MPI function calls. This work is distinguished from DSC [4] in that it provides the ability to interactively create, test and modify a distributed environment using the original interactive language and a portable parallel library. Since MPI has been ported to a wide variety of "supercomputers" and networks of UNIX workstations, largely by layering it on top of existing parallel software packages, any symbolic algebra systems integrating STAR/MPI can immediately take advantage of such parallel architectures. The name STAR/MPI ("$*$/MPI") has been chosen with the idea that "$*$" can be replaced with an arbitrary interactive language. The developers of GAP are cooperating in including GAP/MPI in a future GAP distribution, and GCL/MPI will be contributed to the GCL effort.

Because there is such a variety of parallel systems, it is important to set down specific goals for any new implementation, so as to build up a taxonomy of parallel systems. Perhaps the single most important goal for STAR/MPI was to re-use the existing implementations of parallel systems and non-parallel, interactive languages, rather than "re-invent the wheel". If the advantages of parallel programming are to reach the many implementations of interactive languages, then it is important to develop a methodology that provides parallelism on top of existing software, while requiring little additional developer effort. Section 2 explains the choices made in light of that goal.

Section 4 describes a particularly appropriate class of target algorithms for symbolic algebra. The class appears to parallelize easily under the STAR/MPI model. The same section also defines a property, *weak sequential dependence*, that provides a sufficient condition for an algorithm to be easily and efficiently parallelizable under this model. C libraries are designed with a philosophy quite different from those of interactive languages (see discussion of implementation in section 6 and especially section 6.2). The use of an interactive language raises such issues as command line arguments, handling of signals and interrupts, and treatment of runaway programs. For this reason, and general ease of use, it was felt important to add a higher level interface. The user retains the capability to access MPI directly through the lower layers, but it is anticipated that most use will be through the highest layer.

The highest layer of STAR/MPI (section 3.2) implements a SPMD (single program, multiple data) interface within the context of a master-slave model of computation. The model consists of independently running the same sequential program on master and slave (to guarantee a common

[1] Partially supported by NSF Grant CCR-920449.

to appear, Int. Symp. on Symbolic and Algebraic Computation (ISSAC-95)

environment on all processes) until they reach the function `MasterSlave()`. This function then causes the master to spawn off tasks to be executed by the slaves. All results are returned to the master, and there is a mechanism for the master to broadcast to all slaves any updates based on the results received.

It is envisioned that the majority of sites will run this software on a network of UNIX workstations. Such networks are particularly important in a university environment, where student laboratories with many workstations on a single Ethernet can be used effectively during off-peak periods. Workstations on Ethernet have the disadvantage of high packet latency (many milliseconds), as compared to 10 microseconds or less for many MIMD parallel computers. Yet, their widespread availability makes them attractive for all but the finest grain computations.

It may still be the case that faster processors and lower message latency are needed to tackle certain problems. In such cases, one can take advantage of the portability of MPI and GCL or GAP to easily port to newer, faster architectures, without change to STAR/MPI or to the end-user application code

2 Goals

The goal of STAR/MPI is to integrate parallel programming into existing interactive languages and especially languages for symbolic algebra. This integration should be done in a manner that is easily portable to additional parallel architectures and can be distributed for free. Secondly, STAR/MPI should demonstrate a methodology that can be transferred to additional interactive languages with relatively little development effort. The latter requirement led to looking solely at asynchronous parallelism. In this class, Multilisp [10] is notable as an integration of an interactive language (LISP) on the BBN Butterfly machine, but it does not transfer readily to other parallel architectures.

It was decided to implement STAR/MPI on top of an existing parallel language In order to satisfy the requirement of small development effort. Since many interactive languages are built on top of C, it was decided to choose a parallel version of C as the implementation language. There are several freely distributed parallel enhancements of C. These enhancements fall roughly into two categories: message-based C libraries and enhancements to the C syntax and semantics.

Among the message-based C libraries, there are several notable examples [2, 7, 12]. Our choice of MPI [9, 12] was influenced by its proposal as a portable standard and its appropriately layered feature set. Several developers of other message-based libraries are participating in the development of the MPI standard, and there are several good implementations of MPI on top of previously existing parallel libraries. Admittedly, the choice of MPI is somewhat arbitrary, and PVM [7] also enjoys a great deal of popularity. It is hoped that this work can be ported to other popular parallel libraries as the need arises.

Among the enhancements to the C syntax and semantics, there are also several notable integrations of a non-interactive language (C and C++) with parallel programming [1, 3, 6]. An alternative to MPI would have been to translate the syntax and semantics of a parallel extension of C into the semantics of an existing interactive language on top of C. Such a task presents a dilemma. Translating the syntax of the parallel extensions of C into parallel extensions of the higher level interactive language would likely require much development effort. This violates the requirement of transferring that methodology to additional interactive languages with relatively little development effort. Translating the syntax of the parallel extensions of C into calls to library routines in the higher level interactive language would destroy the elegance of the underlying parallel C. Such a strategy may turn out no better than using an existing message-passing C library.

An additional goal was to present a simple model of parallelism to the user — even at the cost of less generality. It was felt that the current users of symbolic algebra systems have a strong demand for greater CPU power, but little direct experience with parallel programming. Hence it was felt that a simple, but possibly less general, model would be a better fit. This simple model was chosen also with the additional goal of easing the task of parallelizing existing sequential code.

The initial source languages of GAP and GCL were chosen because the complete source code, in C, is freely distributed for both systems. The methodology is intended to extend to the many other interactive languages written in C.

Finally, one hopes to preserve the goals of the underlying parallel library. Among these are preserve portability, heterogeneous architectures, scalability to many processors, and efficiency. Note also that portability necessarily extends only to the intersection of the architectures supported by MPI and the interactive language.

3 Architecture of STAR/MPI

STAR/MPI is implemented in three layers. It is anticipated that most users will use only the highest layer: the master-slave abstraction. The highest layer maintains an easy-to-use SPMD model. The user program appears to run on single (master) process, with occasional tasks spawned to be executed on a slave. The slave tasks are procedures defined on the master and are executed on the slave in the same dynamic and lexical environment as when invoked on the master. The user program is divided into two parts: a *parallel portion* corresponding to invocation of the function `MasterSlave()`; and the remaining portion, known as the *sequential portion*. The sequential portion is executed simultaneously on master and all slaves, thus preserving a common environment. When the parallel portion is reached, the master spawns tasks to be executed by slaves.

The middle layer is the slave-server abstraction. This causes the slaves to become "slave servers", that execute arbitrary commands (and evaluate arbitrary expressions) sent by the master. This appears to be the "lowest common denominator" for those interactive programs that already restrict communication to be between a master and some slaves. The lowest layer is the interface to the MPI library. It is assumed that an end-user would call this lowest layer only when driven to by the need for greater efficiency.

3.1 System user contract

At the highest layer, one can imagine a contract between the system and end-user. The system promises that

- the code never deadlocks,
- the current lexical and dynamic environment is preserved across all processes during the sequential portion of the code,
- there is a barrier to synchronize processes at the beginning and end of each parallel portion of code, and

- there is support for message tracing and performance measurement.

In return, the user is left with the problem of

- choosing an appropriate task decomposition,
- managing a queue of tasks,
- handling all interactions among distinct tasks, and
- maintaining a common dynamic and lexical environment on master and slaves at the end of each parallel portion of code.

The user is responsible for an appropriate task decomposition that results in sufficiently small messages between master and slave so as not to create a communications bottleneck. Further the amount of useful work expected in a task should dominate the overhead of passing a task to a slave and returning a result. In our tests on SPARC-2's over two Ethernet subnets, that overhead time was about 10 milliseconds. It would be less on true parallel computers.

3.2 Master Slave abstraction

The master-slave abstraction, the highest layer, is intended to provide an exceptionally simple user interface. As a SPMD interface, it eases the job of porting existing sequential code to a parallel environment. The master process is the original process started by the user, while the slave processes are those spawned when MPI is initialized. The overriding principle in the design of the master-slave abstraction is to

> *maintain a common program and common lexical and dynamic environment on both master and slave.*

In order to accomplish this, the master and all slaves are initially set to sequentially execute a common program. The parallel portion of the program is executed via a function `MasterSlave()`. This function has as parameters four auxiliary functions: `getTask()`, `doTask()`, `getTaskResult()` and `updateEnvironment()`. All four auxiliary functions must be written by the user. Note that if the auxiliary functions are defined in the same lexical environment as the invocation of the function `MasterSlave()`, then it will have access to all of the variables of that environment. The logic of these auxiliary functions can most easily be understood by following the processing of a single task. Assume a user has written a deterministic program, i.e. one in which the result is independent of the choice of slaves for each task and the time taken for each slave to receive and complete a task. Then the following sequential pseudo-code is functionally equivalent to the actual `MasterSlave()`, which operates on several tasks in parallel. The detailed description follows.

```
function MasterSlave(getTask, doTask,
             getTaskResult, updateEnvironment)
  repeat
    if a slave is free then
      master:  task := getTask();
      if task <> false then
        [label]
        slave:  result := doTask(task);
        master:
        action := getTaskResult(task, result);
        if action == false then done;
        if action == true then
          master and all slaves:
          updateEnvironment(task, result);
        if action == resend then goto [label]
  until task == false and all slaves are free
```

When a slave process becomes available, the master calls `getTask()` to obtain a new task. A numeric tag counter is incremented, and the new task is given that tag. The task is an arbitrarily defined user object. A representation of that object is passed to a slave, along with the task tag. Since MPI supports numeric message tags, the task tag is passed as an MPI message tag.

A slave receives the representation of the new task. The slave then executes `doTask(task)` and returns a result: an arbitrary user-defined object. A representation of the result is passed back to the master, along with the task tag as an MPI message tag.

The master receives the representation of the result and uses the associated tag to retrieve the original task that generated the result. The master then invokes the routine `getTaskResult(task, result)`. One of three values is then returned: `false`, `true` or `resend`. If `false` is returned, no further actions are taken. Since the previous slave process must now be free, a new task is generated by the master, as before.

If `true` is returned, this is understood by the system as a request by the user to update the dynamic or lexical environment on all processes. This causes the user function `updateEnvironment(task, result)` to be executed on the master and all slaves, with the same parameters as for the previous invocation of `getTaskResult()`.

The return value `resend` is understood by the system as a request by the user to re-execute the original task on the same slave as before. This will usually be invoked if the user believes that the dynamic or lexical environment has changed since the original invocation of `doTask()` on the slave process, and that the result of the task is affected by the change in the environment. In that case, the tag counter is incremented, the new tag is associated with the task, and `doTask(task)` is re-executed on the original slave with the same task as before. The rationale for insisting that the execution be carried out on the original slave is that the slave should have the option of caching some partial results and re-using them when it receives the "resend" request. The processing of the task then continues as described above for `doTask(task)`.

It can be a burden on the user to determine if the environment has changed since the original invocation of the routine `doTask(task)`. For this reason, a boolean utility, `IsUpToDate()`, is provided. Whenever the user function `updateEnvironment()` is called, the system saves the current value of the tag counter. If the saved value of the tag counter is larger than the tag of `task`, then the environment is assumed to have changed, and `IsUpToDate()` returns `false`. Otherwise, it returns `true`. A typical usage is for the user to include the statement below within `getTaskResult()`.

```
if not IsUpToDate() then return ``resend''
```

Finally, if `getTask()` should ever return `false`, then the master waits until another result has been returned, and then tries `getTask()` again. If at some point all slaves are free and `getTask()` returns `false`, then execution of `MasterSlave()` completes, and master and slave proceed to execute the main program sequentially until the next occurrence of `MasterSlave()`. If the dynamic and lexical environment are identical at the beginning of a sequential portion of the program, then they will also be identical when the next occurrence of `MasterSlave()` is encountered. It is the responsibility of the user, under the system user contract, to insure that the global and lexical environments remain identical after completion of the invocation of `MasterSlave()`.

3.3 Slave Server

The slave server layer, or middle layer, is based on three system-defined routines:
`SendCommand()`; `GetResult()`; and `BroadcastCommand()`.
The three commands, described below, are all executed on the master process, only.

`SendCommand( command, destination [, opt_tag] )`
The parameter `command` must be a legal statement in the syntax of the interactive language. Since the parameter is evaluated, a literal command must either be quoted (LISP) or put in a string (GAP). The command is then executed on the slave processor with id, `destination`.

`GetResult( [optional_source] , [optional_tag] )`
In MPI, one can restrict to receiving messages from a single source, or messages with a single tag. If the source and/or tag parameters are omitted, they default to the values `MPIany_source` and `MPIany_tag`.

`BroadcastCommand( command )`
This is similar to `SendCommand()`, except that it broadcasts the command to all slaves. The slaves locally execute the command, but they do not return any result. Unlike MPI's broadcast function, `BroadcastCommand()` sends a message only to the slaves, and does not send a copy of the message back to the master.

A subset of utilities from the lowest layer (interface to direct calls to MPI) can also be useful here. Examples include `MPI_Comm_rank()` (MPI process id), `MPI_Status_Source()` (source of last message), `MPI_Status_Tag()` (tag of last message), `MPI_Comm_Size()` (number of MPI processes), and `MPI_Iprobe()` (non-blocking test for new messages).

3.4 Direct Calls to MPI

At the lowest level, one can make direct calls to the system services provided by MPI. Only the point-to-point and environment layers have been implemented, since that is all that was needed to support the higher layers. By following the existing pattern, it is easy, but tedious, to implement the remaining MPI library functions.

The original MPI library calls typically have many parameters. In keeping with the philosophy of interactive languages, the number of parameters was reduced and simplified. One decision was to replace some trailing arguments to MPI calls by optional arguments with reasonable defaults. It was also decided to remove all arguments that referred to a communicator, and always assume the single MPI communicator, `MPI_COMM_WORLD`. Further, a common argument pattern was (buf, count, datatype) or (buf, count). Such patterns were replaced by a single string argument, with the datatype being character. Thus, all messages passed within STAR/MPI are assumed to be strings. General data objects can still be passed by first converting to strings in a marshalling process described in section 6.1.

4 Target Algorithms

While many sequential algorithms do not readily parallelize, one class of algorithms with considerable importance in symbolic algebra does parallelize easily. This is the class of algorithms with the *weak sequential dependence property,* defined below.

The purpose of the algorithm is to construct a global data structure with certain closure properties. It is done incrementally by extracting pairs of objects from the data structure and testing them against the global data structure. Most pairs yield no new information. Some pairs modify the data structure. The modified data structure yields new pairs to test. The algorithm continues until no pairs yield new information, at which point one has *closure* or *termination.* In addition, the algorithm must satisfy a *monotonicity condition.* If a pair yields no new information in the current data structure, then it will yield no new information in all future data structures. The pairs are often called *critical pairs.*

Any algorithm satisfying the above properties is said to have the weak sequential dependence property. Some examples of this class of algorithms are Groebner bases, Knuth-Bendix, LLL (with applications to polynomial factorization), coset enumeration, resolution theorem proving, and strong generating sets. The first four algorithms can be found in [15], while the last two originally appeared in [13] and [16].

An algorithm with such properties can be written in STAR/MPI by the user by defining the four auxiliary functions to `MasterSlave()` as follows.

`getTask()`: Get the next critical pair from the current data structure.

`doTask(crit_pair)`: Test critical pair against the current global data structure.

```
GetResult(crit_pair, result)
  if result == false then return false
  else if IsUpToDate() == true then return true
  else return 'resend'
```

`updateEnvironment(crit_pair, result)`: Apply new information in `result` to update global data structure.

5 Writing Code in STAR/MPI

5.1 Example: factorization in parallel

The following example would be defined in a file that is loaded into the master and all slaves. The function below, `myfactor(number)`, would then be called on master and slaves at the same time. A standard sieve algorithm is used. The master assigns 1000 consecutive numbers to a slave to test for the presence of factors. It returns a list of factors and their exponents. For example, `myfactor(12)` returns `[[2,2],[3,1]]` ($2^2 \cdot 3^1 = 12$).

```
function myfactor (num)
  sqrt := ceiling(sqrt(num));
  currNum := 1;
  answer := [];

  MasterSlave(
    getTask:
    function()
      ifcurrNum < sqrt then
        currNum := currNum+1;
        return currNum;
    end function,
    doTask:
```

```
    function(tstNum)
      for j from tstNum to tstNum+1000 do
        ifnum mod j = 0 then
          Set exp := maximum integer e
            such that num mod jê = 0;
          return [j,exp];
      return false;
    end function,
    getTaskResult:
    function(result)
      if result <> false then
        if result not in answer then
          Add result to answer;
          currNum := result[1] + 1;
      return false;
    end function
    );
  return answer;
end function;
```

5.2 Code developement

It is intended that code development and code parallelization be relatively easy. If one already has working sequential code, then this can be used as the basis for the definition of the four auxiliary functions of section 4. In any case, one's intuition in the sequential case will carry over to the parallel case.

The next step is to determine the loop within which a sequential program would make repeated calls to the four auxiliary functions. This loop should be replaced by a call to `MasterSlave()`. The four auxiliary functions should be defined in the same lexical environment as the invocation of `MasterSlave()`.

For purposes of debugging, it is worth writing an auxiliary file that defines a sequential version of `MasterSlave()` equivalently to the pseudo-code of section 3.2. One can then load the auxiliary file and test if the program performs correctly in a sequential environment. This provides a necessary condition for successful parallel execution.

The debugging strategy above fails to simulate an important aspect of the parallel environment. That fault can be partially remedied at the cost of some additional effort. In the parallel environment, no communication between master and slave should take place, except when the result of `doTask()` is passed on to `getTaskResult()` and to `updateEnvironment()`. This can be simulated in a sequential environment by the following expedient. Before `MasterSlave()` is called, a copy is made of any variables used by `doTask()` but outside its lexical scope. The copies are initialized to the values of the original variables. Additional formal parameters are added to `doTask()` with the names of the original variables (not the copies). When `doTask()` is called, the formal parameters are bound to the copied variables. Further, `updateEnvironment()` is temporarily modified to update both the original variables and the copied variables.

Finally, the global variable `trace` causes the task and result parameters to be displayed on the screen when `trace` is `true`. Its default value is `false`.

6 Implementation

6.1 Implementation Requirements

The implementation strategy is critical to making this work portable to other interactive languages. The requirements for porting to a new interactive language are listed below. Of these, the first two are absolutely required. The next three are not required, but make it easier for the end user to develop and debug in a parallel environment. The last one may or may not be an issue, depending on the language's internal implementation. The requirements are:

- a foreign function interface or source code for the interactive language;
- a marshalling mechanism and perhaps eval;
- a user-defined error handler;
- a user-defined interrupt handler;
- definition of functions with access to non-local, non-global, lexically scoped variables;
- and maintaining a contiguous, internal heap.

Marshalling is a mechanism to convert a general data object into a linear representation suitable for the body of an inter-process message. Demarshalling is the inverse operation. This is required to pass arbitrary language objects between master and slave. In an interactive language, it is also desirable to be able to marshal general commands of the language, so that the master can be directed by the user at runtime to issue a new command not foreseen in the context of the initial program loaded by master and slave. Of course, if commands are to be passed, then a user-function, eval, is also needed. In LISP, the additional requirement is trivial, since eval is provided, and commands are stored in the same manner as a list data structure.

Most interactive languages do not have a built-in marshalling mechanism. In contrast, Concert C [1] is a parallel language where they have gone to great lengths to build an efficient marshalling mechanism. Nevertheless, the functions, read and print, are sufficient to build a "poor man's" (less efficient) marshalling mechanism. Some of this inefficiency could be eliminated in future versions by incorporating work on marshalling for symbolic algebra systems, such as the work on MP [8]. Many interactive languages, such as LISP, do provide the user functions, read, eval, and print. In GAP, read, eval and print are not provided as end-user functions. However, the source code is provided free, and so it was not excessively difficult to extract the C functions that served that purpose, and wrap some code around them to make them available as end-user functions. This, of course, required re-compiling the kernel.

A user-defined error handler is also needed. If an error occurs on the master, the original interactive language can probably be made to recover cleanly. But if an error occurs on the slave, the slave is typically thrown into a break or error loop. Since the user has no access to the standard input of the slave process, this presents a dilemma. The solution chosen was to add an appropriate interrupt handler to the slave processes.

The GCL error handler is imbedded in the LISP code for its read-eval-print loop. Although the error handler is not documented, the code is clear and there are enough hooks to easily implement a user-defined error handler. GAP does not provide a user-level error handler. Extracting one from the C source code required understanding GAP's internal stack and its use of setjmp. GAP's error handler is not currently documented. The GAP kernel was modified to intercept calls by a slave GAP process to the GAP error handler. This currently works, but it is planned to talk with the GAP developers about other strategies that may be more maintainable across new versions of GAP.

It is somewhat surprising that a user-defined interrupt handler was needed. This is because we were using an early MPICH [5] implementation of MPI, which used rsh to spawn slaves, and rsh propagates some of its signals (SIGINT, SIGQUIT and SIGTERM). Hence, an interrupt within the master was found to affect the slaves. Later versions of MPICH also offer a server mechanism and a more customized mechanism, which avoid the use of rsh in this context. However, the MPI standard does not address the issue of propagation of signals, and so this can also be an issue for other versions of MPI.

A related issue was what to do when evaluation on a slave resulted in an error. It was decided to propagate the error back to the master as a "result". Thus, one has the unusual situation of a command by the master that has generated an error on the slave, but the effect is delayed until the master gets the result from the slave.

The last implementation requirement is really a condition of the implementation of the interactive language. MPI must allocate memory for buffers and the interactive language must allocate memory for the heap. Naturally, any conflicting usage of malloc and sbrk among the two systems must be resolved. But there is a deeper issue.

Some implementations of an interactive language expect their heap to grow in a contiguous region as more memory is required by the user program. This requirement can be violated if the interactive program initializes its heap at startup time, then calls MPI's initialization routine, and then the user program requires more heap memory. The issue can be complicated further if a growing queue of pending messages forces MPI to later allocate additional memory for buffers. This latter case has been carefully avoided in the highest layer of STAR/MPI by limiting the depth of the message queue.

Nevertheless, the requirement of a contiguous heap remains an issue for STAR/MPI. Otherwise, a user program may suddenly crash when it needs additional memory. Steve Linton brought out this issue and also suggested a solution that works well when MPI uses malloc and the interactive language uses sbrk. This is the case for the two implementations of STAR/MPI considered here. The startup routine of the interactive language must be modified to initially call malloc to allocate a buffer of sufficient size to hold all MPI buffers that might later need to be allocated. Generally, this size is proportional to the number of slaves being allocated. The malloc buffer should then be immediately released using free, thus placing that memory in the malloc arena to later be re-used by MPI. Having done this, the startup routine can then set up contiguous heap in high memory, using the original code involving sbrk.

6.2 Surprises

Since the MPI library was originally designed purely for C and FORTRAN, there were several surprises in adapting it to an interactive environment. Porting MPI to an interactive language necessitated considering such "new" features as read-eval-print loops, break or debug loops, signal handlers for user interrupts, error handlers (usually invoked through the C library routine, setjmp), command line parameters, timeout criteria for infinite loops, environment variables (e.g. `$PWD`), new data types (string, symbol, interpreted code), dynamic object creation and garbage collection, and a potentially growing, contiguous internal heap.

Some of these issues have been discussed in section 6.1. The surprises discussed here (along with the issue of interrupt handlers of section refrequirements) largely relate to the MPICH [5] implementation of MPI. In particular, most of these surprises are a consequence of the decision to inherit the p4 [2] mechanism for spawning slaves through the UNIX utility rsh. Other MPI implementation strategies are likely to result in different surprises.

MPICH adds several undocumented command line parameters when it spawns a slave via rsh. Initially, this caused some confusion in the implementation of STAR/MPI, until it was realized that command line parameters meant for MPICH were being intercepted by the interactive language.

A timeout mechanism was needed for runaway slaves, since it can be quite laborious to manually kill processes on each of many slave processors. This resulted in a surprising interaction with rsh. If rsh spawns a shell that already has a timeout mechanism, then that timeout mechanism may be invoked even before the user-specified timeout takes effect.

The issue of limited swap space is affected by that of timeouts. If a user specifies a timeout of 60 minutes, and during debugging is forced to kill the master process, then the slave processes continue for 60 minutes. If this is repeated 10 times in an hour, then one will occupy 10 times as much swap space. Where swap space is limited, this can result in obscure bugs as some slave processes are correctly spawned, but not others. Presumably this could be fixed by the way in which MPICH spawns slave processes through rsh. As a hedge, the user should adjust the interactive language to not use more virtual memory than the job needs.

6.3 Implementation Experience

The MPICH implementation [5] of MPI was integrated into both GCL and GAP. The time required for each language was about one 40-hour person-week. GCL was attempted first, and much of the time spent on GCL/MPI was needed to learn the finer issues involved in integrating a parallel interface and debugging the LISP code. The little new C code needed for GCL/MPI was required largely to convert from MPI/C conventions to LISP conventions for the lowest layer, described in section 3.4. A LISP macro was found extremely useful in automating the LISP/C interface, so that some MPI library routines in C could be imported into LISP using this macro, while writing no C code at all.

The time for GAP/MPI was spent differently. Because GAP is a newer language, it does not yet offer to end-users all of the language features bulleted in section 6.1, and which exist in GCL. For example, since GAP has no foreign function interface, one must modify the kernel to make new C functions available as end-user functions. Again, C macros were found useful in automating some of this work, although there remained significant C code to write beyond what was provided by the C macros. It's expected that in the next distribution (GAP 4.1) many of the internal features will be included as end-user functions. In any case, because the source code for the GAP kernel is freely available, this lack of end-user features was not a significant barrier to integrating MPI.

The number of lines of code was 379/537 for the MPI layer, 107/105 for the slave listener, and 173/181 for the master-slave layer. The first number is always the number of lines of GCL/C code, and the second number is always the number of lines of GAP/C code.

7 Conclusion

STAR/MPI has been developed as a general architecture and tested in separate bindings to both GAP and GCL. There have already been some successful user programs developed in GAP/MPI and additional people have expressed an interest in testing the system. In the completed experiments, up to 25 SPARC-2 CPU's were attached to gain additional CPU power. The tests were on tasks which were relatively straightforward to parallelize. There was almost complete CPU utilization during these experiments, but testing on still larger examples will be needed for more quantitative conclusions.

When tasks are partitioned in sufficiently large chunks, the overhead of communication is not a problem. However, for more fine-grained computations, one would envision extending this work to integrate some of the previous work on marshalling mechanisms and distributed symbolic computation in general [8, 11]. Another possible direction for future work would be heterogeneous, distributed computation. With an appropriate marshalling mechanism, one can eliminate the restriction that all processes execute the same language. Indeed, some of the slaves might potentially be a non-interactive and/or numerical language.

Finally, the initial implementations of STAR/MPI are on networks of workstations, which is fundamentally a distributed architecture. Since MPI (and other parallel libraries) run on other classes of parallel computers, it should be feasible to port this architecture to such other architectures. The largest research issue in such a port is probably the issue of garbage collection on shared memory machines, since one processor can disturb an ongoing computation of another. Thus, one would need to invoke a barrier followed by a mechanism for each processor to modify its pointers.

GAP/MPI will be included as part of the next distribution of GAP. It is also planned to offer GCL/MPI to the users of GCL. Through the experiences of others and the author concerning end-user applications, it is hoped to gain greater feedback on the suitability of this model for typical CPU-intensive problems that are encountered.

Both GAP/MPI and GCL/MPI are freely available on request from the author at `gene@ccs.neu.edu`. It's expected that GCL/MPI can be ported to other dialects of LISP without too much effort. GAP (and its 1083 page manual) are available by anonymous ftp from ftp.math.rwth-aachen.de, dimacs.rutgers.edu and pell.anu.edu.au (all at /pub/gap), among other sites. GCL is available by anonymous ftp from math.utexas.edu and ftp.cli.com in /pub/gcl/gcl-XXX.tgz. MPICH (a MPI implementation) is available by anonymous ftp from info.mcs.anl.gov:/pub/mpi/mpichXXX.tar.gZ.

8 Acknowledgements

The author thanks Ivan Judson for his initial explorations of MPI and other parallel systems. He also 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. Finally, the author acknowledges the many valuable comments by the above, William Clinger and the referees, which have helped to make this a better paper.

References

[1] J. Auerbach, D. Bacon, A. Goldberg, G. Goldszmidt, M. Kennedy, A. Lowry, J. Russell, W. Silverman, R. Strom, D. Yellin and S. Yemini, "High-Level Language Support for Programming Distributed Systems", *International Conference on Computer Languages* (1992).

[2] R.M. Butler and E.L. Lusk, "Monitors, messages, and clusters: The p4 parallel programming system", *Parallel Computing* **20**, (1994) pp. 547–564.

[3] K.M. Chandy and C. Kesselman, "Compositional C++: Compositional Parallel Programming", Proceedings of the Fourth Workshop on Parallel Computing and Compilers, Springer-Verlag, to appear.

[4] A. Diaz, E. Kaltofen, K. Schmitz and T. Valente, "A System for Distributed Symbolic Computation", *Proc. of Int. Symp. on Symbolic and Algebraic Computation (ISSAC-91)*, ACM Press, 1991, pp. 323–332.

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

[6] T. von Eicken, D.E. Culler, S.C. Goldstein and K.E. Schauser, "Active Messages: a Mechanism for Integrated Communication and Computation", Proceedings of the International Symposium on Computer Architecture, 1992.

[7] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam, *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*, MIT Press, 1994.

[8] S. Gray, N. Kajler and P. Wang, "MP: A protocol for efficient exchange of mathematical expressions", *Proc. of Int. Symp. on Symbolic and Algebraic Computation (ISSAC-94)*, ACM Press, 1994, pp. 330-335.

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

[10] R.H. Halstead, Jr., "Multilisp: A Language for Concurrent Symbolic Computation", *ACM Transactions on Programming Languages and Systems* **7**(4), 1985, pp. 501–538.

[11] N. Kajler, "CAS/PI: a Portable and Extensible Interface for Computer Algebra Systems", *Proc. of Int. Symp. on Symbolic and Algebraic Computation (ISSAC-92)*, ACM Press, 1992.

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

[13] J.A. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle", *J. of the Association for Computing Machinery* **12**, 1965.

[14] M. Schönert et al., GAP – *Groups, Algorithms and Programming* (Manual), Lehrstuhl D für Mathematik, RWTH, Aachen, Germany, 1993.

[15] C.C. Sims, *Computation with finitely presented groups*, Cambridge University Press, 1994.

[16] 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.