MARS: RUNTIME SUPPORT FOR COORDINATED APPLICATIONS

Neal Sample **Carl Bartlett** **Matthew Haines**

Department of Computer Science
University of Wyoming
Laramie, WY 82070
USA

{nsample, bartlett, haines}@cs.uwyo.edu

Keywords: multidisciplinary application; runtime system; legacy code; MARS.

ABSTRACT

Expert knowledge from many disciplines is frequently embodied in stand-alone codes used to solve particular problems. Codes from various disciplines can be composed into cooperative ensembles that can answer questions larger than any solitary code can. These multi-code compositions are called *multidisciplinary applications* and are a growing area of research. To support the integration of existing codes into multidisciplinary applications, we have constructed the Multidisciplinary Application Runtime System (MARS). MARS supports legacy modules, heterogeneous execution environments, conditional execution flows, dynamic module invocation and realignment, runtime binding of output data paths, and a simple specification language to script module actions.

PROBLEM

As expert knowledge is more and more frequently embodied in computer models and simulations, the opportunity to compose useful models from multiple disciplines is increasing as well. In order to better solve many complex problems, it is useful to draw together domain expertise from multiple disciplines.

For instance, designing an automobile requires information about structure, propulsion, guidance, safety, and many other features. Designs are done in teams, with each team member (or group) having specific tasks, completing one part of the puzzle. The group responsible for a new engine design is not the same group responsible for the body design. Further up the scale, the entire design team could be seen as one small component themselves, coupled with advertising, sales and manufacturing teams to deliver the product to market. The combination of these different expert groups can achieve much more than any group by itself.

The automobile production problem is analogous to what is occurring in scientific research today. Models are taken from different disciplines and combined in innovative ways to provide answers that the component models could not provide by themselves. The problem of combining models from more than one discipline is known as the *multidisciplinary optimization problem*, and the resulting applications are called multidisciplinary applications.

The general MDO construction problem has three main components: managing concurrency and parallelism, data exchange, and basic control functionality (such as module invocation and termination). When using heterogeneous codes, little can be done to combine them (without rewriting the working modules to be compliant with the flavor of the month task and data parallel language system); there are very narrow design paths MDO builders usually follow. The path to managing concurrency problems is often to serialize the component processes, thus ensuring consistency at a cost of efficiency. The path for data exchange is commonly through UNIX files. Basic functions, like invocations, can be done with scripting languages or with system calls, such as the exec() in UNIX.

An important consideration in the development of MDO codes is that desirable component modules often exist, but were not written to be used in a cooperative environment. These *legacy codes*, while capable of performing a desirable task, are often limited in use because they are difficult to tie together with other legacy codes. Even MDOs built with newer modules designed with interfaces that are mindful of *specific* legacy codes are ineffective. Interfaces to the resulting new MDO collections are often as idiosyncratic as the legacy code that is the basis of the MDO; it is therefore difficult to expand or interchange other modules with the new MDO. For instance, increasing the variety of *Phillips* screws in a package does not make that package work better with a *flathead* screwdriver. In essence, to use legacy codes, designers must completely build around them, suffering the costs incurred when coupling heterogeneous modules, or rewrite them to comply with local modules, duplicating significant effort already expended on the legacy code's original design.

It was with these many considerations (data exchange, concurrency and parallelism, basic functionality of a component system, and support for legacy codes) in mind that we built *the Multidisciplinary Application Runtime System* (MARS). MARS supports abstract data passing, asynchronous partially-ordered message passing, remote service requests, dynamic module invocation, and various primitives for module control. As such, it could serve as the runtime support for another task parallel high-level language specification, like Opus (which was designed for building MDO codes) [2, 3, 5].

MARS also includes components beyond the runtime system proper. To fully support the construction of MDO codes, MARS understands a simple specification language that defines the *working modules* (or *tasks*) within an MDO. A collection of working modules and a *leader* (specification) to guide those modules are referred to as a MARS *ensemble*.

MARS also has a server component that resides on all execution nodes (CPUs) that participate in a running ensemble. This server, called an *mdo_server*, functions to invoke working modules on specific execution nodes at the request of another module. The components of a leader (and how sub-leaders and working modules are added to an ensemble that follows the leader) are also part of the specification language.

To support legacy modules, MARS includes a source to source compiler called the wrapper. The wrapper takes source for a working module, wraps all the I/O routines with MARS calls, and augments the original code to act as persistent server capable of processing MARS messages. This wrapper is *not* dependent on user augmentation of original source, as with a full language specification like Opus. Without any annotation of the source by the MDO builder, the MARS wrapper replaces the legacy code's original entry point with MARS module server code, while preserving access to the original entry point for later use.

A powerful module composition tool is needed to bring tools from various disciplines together to form multidisciplinary applications. When legacy modules are developed, their design generally does not consider how the module could be used as part of a future collections of modules. Legacy codes are frequently designed to achieve a single task in the best possible way. As such, whatever local tools, methods, and protocols are available are used to build the best possible *stand-alone* module. This local construction process makes it difficult to integrate modules from different disciplines when they do not use common data formats, common communication protocols, or even common development languages. MARS is a composition tool designed to bring modules together from multiple disciplines to form multidisciplinary applications.

RELATED WORK

There have been many different approaches to achieving task integration. Most approaches deal with homogenous tasks rather than integrating codes from multiple sources. We have found that similarities exist between elements of particular systems and certain MARS components, but that the collected functionality of MARS has not been realized.

A primary difference between MARS and virtually all languages and language extensions is the support for legacy codes. The purpose of language specifications is generally to build *new* codes, including *new* MDOs. They are designed with little support for legacy modules in mind. Limited code reuse (using legacy modules) can be achieved with these language-level specifications, but only by reusing other codes originally written using a specific language specification.

It should be further noted that many language specifications could be compiled to MARS, although MARS currently lacks a native threading system. (The initial focus of the MARS project was to develop the message passing layer, the I/O routines, and the basic control and concurrency primitives. The assumption is that MARS will be augmented with an existing threads package.)

Fx

Developed at CMU, Fx is an extension of the Fortran language. In Fx, parallel sections of code, called tasks, are invoked by a program as subroutines. Tasks communicate with one another by sharing input and output arguments in their *parent* space. The data flows are static and cannot be changed at runtime. This method does have the advantage of possible compile-time path optimizations.

While good for the design of new task parallel codes, Fx is not particularly suited for MDO applications [2]. First, within a single invocation, the Fx language provides excellent support for communication and parallelism. However, between two distinct codes, both written in Fx, the MDO builder is once again responsible for handling communication. In effect, a *static* (where modules are never added or removed) MDO could be easily built as a single Fx program, but altering the ensemble would require re-coding and recompiling the entire Fx code (or else the user has to build a runtime system to provide the needed support).

The Fx *input* and *output* directives radically limit the usefulness of tasks in a parallel environment [2]. If tasks must communicate frequently, routines must be frequently split at synchronization points so the task can share its data. This of course makes programming the tasks difficult and greatly increases the overhead of task invocation.

Orca

Orca is a task parallel language that supports process allocation in a manner similar to MARS. Like MARS module invocations, Orca tasks can be dynamically created and mapped to specific execution nodes. However, there is no explicit message passing in Orca. Instead, communication is handled by applying user defined Abstract Data Type (ADT) operations on *shared objects*. Orca collections are good for coarse-grained parallelism as operations on ADTs are always exclusive and indivisible [2].

Once again, since Orca is a complete programming language, users are limited to programming in Orca. This does limit the possibility of using legacy codes, and it also prevents users from easily extending the Orca specification. Because MARS concentrates on coordination rather than programming, it has a broader range of usefulness to legacy codes.

Additionally, because Orca is compiled and has its own runtime system, users may find it difficult to add to the Orca system. Since MARS is built in clearly defined, user-accessible layers, inserting functionality at the various levels is much simpler than with Orca. Finally, extending higher-level MARS constructs with source language (such a C) primitives is also possible, whereas there is no such distinction with Orca.

Opus

The Opus language specification was designed for multidisciplinary applications [2]. Opus takes a data-centric approach to the MDO problem, introducing the *ShareD Abstraction* (SDA) for coordination and communication. SDAs act as data repositories, communication channels, and occasionally as compute nodes within Opus ensembles.

The Opus language extends HPF with its task parallel programming constructs [3, 7]. There are several strong similarities between MARS and Opus. (This is to be expected; runtime support for an Opus-like language was the genesis for the MARS project.) Multidisciplinary Opus applications begin with a "leader" that sets up the SDAs, similar to the way MARS ensembles are setup. Opus and MARS both use similar *spawn* operations to invoke working modules with specific resources. Opus provides access to SDA objects without working modules having to explicitly locate the SDA objects on the network. Similarly, the leader module in a MARS ensemble defines I/O paths for working modules at runtime.

Opus is a coordination language, not a distributed programming language like Orca or Fx, which makes it more closely related to MARS [3,4]. The MARS runtime system is predicated on primitives exclusively designed for coordinating modules, rather than programming modules.

CHAIMS

The CHAIMS system is similar to MARS, but operates at a significantly higher level of abstraction than MARS. CHAIMS, unlike many of the other coordination systems examined, is close to being a pure composition language, functioning at a higher level of abstraction than task-parallel programming languages and extensions [8,10].

The original CHAIMS specification was for a composition language that would be used to build *megaprograms*. *Megaprogramming* is a form of *programming in the large*, where "large" can be viewed on several independent levels (largeness in time (persistence), large variability (diversity), largeness in size (complexity of the program and/or its range of applicability), and largeness in capital investment (infrastructure)) [11]. Megaprograms compose *megamodules* into collections capable of greater functionality than an individual megamodule can achieve. A megamodule, as originally specified, was a large, persistent, self-contained object with a consistent interface for querying.

CHAIMS assumes that megamodules have their own ontology and maintenance, and are simply available to the public for access. In this sense, megamodules are much like object resources in CORBA. MARS is perhaps better characterized as "programming in the small" in this regard. MARS ensembles are formed by collecting stand-alone modules that are not already servers and turning them into servers capable of being composed into ensembles (megaprograms).

CHAIMS assumes that composers (megaprogrammers) generally do not have control over the megamodules they use [11]. MARS assumes that, in general, its modules may be invoked on any participating execution node, and is thus steered toward a different set of problems. At the same time, however, MARS can certainly communicate with persistent modules; MARS simply extends the CHAIMS specification to include dynamic module invocation primitives. This is a clear philosophical difference: CHAIMS is not "polluted" with non-compositional features, while MARS is not "limited" to dealing with persistent modules.

MARS is a *runtime system* that provides specific data transport mechanisms to multidisciplinary programs and a simple specification language for composition of those programs. CHAIMS is predicated on communication with existing modules through various methods, including COM, CORBA, Java-RMI, etc. [9,10]. CHAIMS does not provide data transport mechanisms to previously naïve programs; rather it has a rich set of messaging protocols intended for communicating with existing network-aware servers (megamodules).

The combination of these implementation differences distinguish the MARS problem set from CHAIMS'. MARS is intended for use primarily with local modules. These local modules are controlled by ensemble creators. They can be altered locally and have multiple invocations on arbitrary execution nodes. CHAIMS assumes that megamodules are available somewhere, and that megaprogrammers have no control over megamodules and cannot alter them.

THE MARS SYSTEM

The MARS system consists of six main interacting components, the wrapper code, the native communications layer, the Remote Service Layer (RSL), the Remote Service Spawn (RSS), the message processing layer, and the MdoModule layer. There are four main divisions of these components: preprocessing components, runtime library code, support for remote invocation, and high-level objects needed to construct the leader code. The components combine to form a complete framework capable of supporting the multidisciplinary problems outlined in this paper. In general, the MARS system works as follows:

1. Component modules necessary to solve a problem are assembled.
2. The modules are wrapped, turning them into specialized MARS servers.
3. A leader is written (the leader is an "executable map" of the execution and data paths to be taken by the modules.)
4. The leader is executed, causing execution of the component modules in accordance with the specification.

When assembled, the leader and the working modules form a multidisciplinary application. This section reviews the components of the MARS system.

The wrapper is the first part of the MARS system. Its job is to turn legacy modules into special MARS servers, capable of communicating with other MARS working modules and leaders. The wrapper installs communication mechanisms in legacy codes and also substitutes MARS I/O routines for those in the original code. Once this is done, a MARS leader code (which is really just a specification for a set of modules) can control the working module by invoking it with the remote invocation tools. Once a module is running, it communicates with its leader through several layers designed to ensure consistent communication between modules. These components relate in the manner shown in Figure 1.

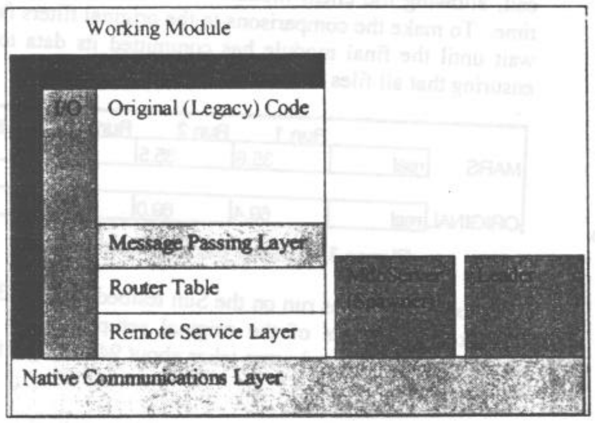

Figure 1. MARS component interactions

<u>File (I/O) Layer</u>

The critical issue with the wrapper is I/O. Many potential candidates to become modules in an MDO use arbitrary output schemes that use multiple files, in multiple (often proprietary) formats that are without obvious meaning to an outside observer. Such programs can be extremely difficult for a developer to understand and integrate into a multidisciplinary optimization. The wrapper must be able to convert simple I/O calls automatically to some other form that can be utilized in a collection of different modules. Disk-based I/O routines are by far the most common, especially in scientific codes. Simulation data is generally pushed to disk and later analyzed or used by other programs.

<u>Remote Service Layer (RSL)</u>

The Remote Service Layer (RSL) is the primary method of communication between modules. RSL communication is ordered between modules along virtual connections because the native communications layer is similarly ordered. RSL messages are asynchronously transmitted and received, without explicit return values in the general case. Special fields in an RSL message header facilitate synchronicity (if desired), but this synchronicity comes from how the message is processed in the Message Processing Layer. RSL messages consist of two parts: a well defined header and a corresponding binary data-blob sent immediately after the header.

<u>Router Table</u>

The Router Table is an advanced data structure for installing arbitrary names in a module's I/O routines. The Router Table provides a bridge between the Remote Service Layer and File I/O Layer. An unaltered, non-wrapped module may expect to receive incoming data from a file, say "`foo`". Instead of demanding that the wrapped module constantly write to a fixed channel "`foo`" (which could go to disk or network), the Router Table allows the destination "`foo`" to be changed dynamically, unbeknownst to the module using "`foo`". This ability is extremely powerful. Unlike the languages using compiled communication paths, like Fx, MARS modules never need to know where their inputs come from until after they are invoked. Likewise, their outputs can be similarly assigned at runtime and can be changed multiple times during their execution lifetime.

<u>Message Processing Layer (MPL)</u>

The Message Processing Layer is the most critical component of the MARS system. It is the layer that translates and implements all requests in all RSL messages. It modifies connection tables, invokes native procedures within wrapped modules, and establishes data connections.

Most mundane functions are also performed by the Message Processing Layer. Log files are opened and written to by the MPL. The level of logging to be performed by the lower layers (such as File I/O Layer logging) are altered with calls to the MPL. It is the layer that understands how to control all the other layers and the only level at which modules can communicate directly.

APPLICATIONS/RESULTS

The section demonstrates three test ensembles constructed with MARS. The first code presents MARS used to implement a pipeline parallelism example. Three graphics filters are applied, in succession, to a series of images. The three filters, which increase sharpness (*is*) in an image, posterize (*po*) an image, and reduce noise (*rn*) in an image, are wrapped to become modules and then fed multiple files from a leader code. The example shows the ease of constructing what could otherwise be a complex pipeline and how that pipeline can dramatically improve performance.

The second example uses the identical wrapped modules from the first example in an advanced arrangement to greatly enhance the performance of the graphics filters. This extended pipeline parallelism test demonstrates how MARS can be extended to include dynamically module invocation to overcome performance bottlenecks. This example shows how ensembles can be easily rearranged without re-coding them.

The final example uses an actual scientific research code taken from the biological sciences. In this example, we demonstrate dramatic performance increases using our system. This test ensemble, used in actual research, is a litmus test for the applicability and performance of MARS on legacy modules. The code is multidisciplinary, and all of the modules were designed before the conception of MARS.

Pipeline Parallelism Example

This example will show how the MARS system was used to wrap three similar codes into an efficient parallel image processing system. We will use three graphics filters and turn them into MARS modules. As noted earlier, the three filters increase sharpness (*is*), posterize (*po*), and remove noise (*rn*) from 8-bit bitmap images.

This experiment uses 40 8-bit black and white images, each 256x256 pixels in size. The images are fed in succession to the three filters. The three filter modules are combined to achieve image compression by bit reduction. The first module, *is*, is applied to refine an images' edges. The second module, *po*, reduces the images' bit depth, using an un-weighted naive heuristic. The third module, *rn*, reduces excessive noise by looking at disparity in neighboring byte values, produced by the bit reduction step.

These three modules are composed with a leader code to produce the ensemble shown in Figure 2.

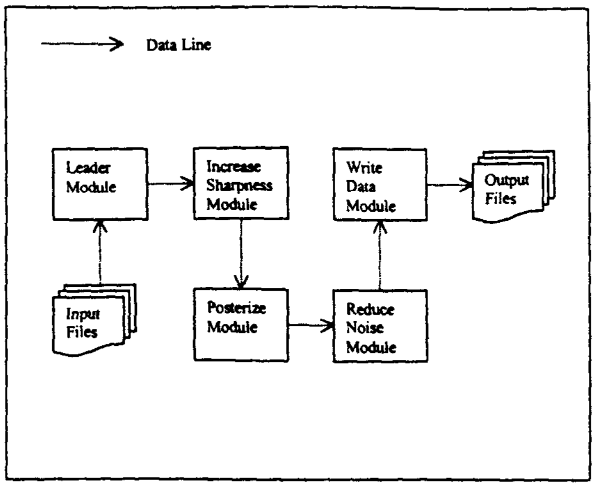

Figure 2. Pipeline parallelism example

We also add a fourth module, one written to handle the disk I/O at the end of the execution path. It would be simple enough to allow the reduce noise module to write the images to disk, but this additional helper module's function will become apparent in the next example.

Most of the leader code is devoted to *locating* the working modules on the network. A significant portion of code is also used to read data into the leader. As such, the leader will become an active participant in this collection of modules. Better performance could be achieved if the increase sharpness module read in the data, as it would not have to be passed from the leader to that module. Since the data files are only 65536 bytes (64k), the overhead of having the leader pass the data an extra time is not too significant.

The last call the leader makes is `leader->end_reply()`. This is a synchronous routine. The `leader->end_reply()` waits for all client modules to terminate before the leader exits. As such, we could increase performance by using an asynchronous `leader->end()` call, allowing the client modules to terminate on their own time. To make the comparisons to the original filters fair, we wait until the final module has committed its data to disk, ensuring that all files are stable.

		Run 1	Run 2	Run 3	Average
MARS	real	35.6	35.5	35.9	35.67
ORIGINAL	real	69.4	69.0	69.5	69.30

Figure 3. Pipeline parallelism results

This experiment was run on the Sun testbed. Figure 3 shows the execution times of the original setup and the MARS modules. The original setup takes about 94% longer than the MARS setup.

With a three stage pipeline, a two-thirds reduction in time would be expected. In this case there was only a one-half reduction in time. With a bit of module profiling, the reason became clear. The increase sharpness module and the reduce noise module take significantly more computation time than the posterizing module. Since those two stages dominate the overall process time, a simple pipeline could only hope to double performance.

With the groundwork for this set of modules in place, we will now demonstrate additional features of MARS that facilitate extending existing setups.

Extended Pipeline Parallelism Example

After improving the performance of the original graphics modules by pipelining and then profiling the resulting collection, it became obvious that a different setup was required for optimal performance. Since the first and third stages (*is* and *rn*) of the overall process dominated the execution times, we multiplied the number of execution nodes at those stages. Figure 4 shows the new module configuration.

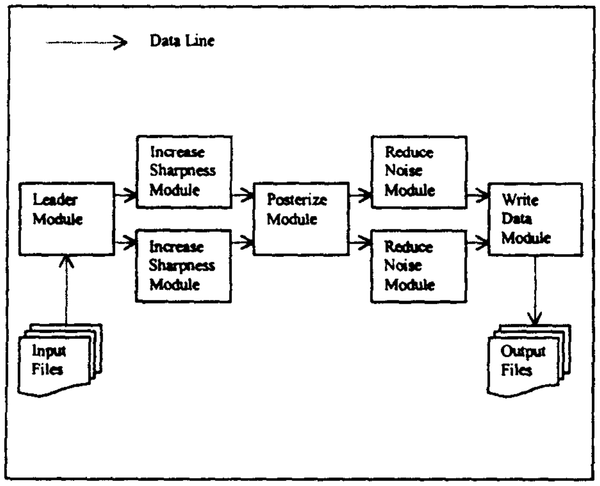

Figure 4. Partial 2-way pipeline

With MARS, moving from the arrangement in Figure 2 to the arrangement in Figure 4 is simple. The leader code remains virtually unchanged, except for the actual control loop. Two new module (one each, *is* and *rn*) specifications are added to the leader, and the leader also adds the required data paths. The control loop in the leader is almost identical, with the data read-in passed alternately to the two increase sharpness modules.

To execute the new MARS module collection efficiently, more CPUs were required. To handle the extra load, the leader and write-out modules were placed on SGIs workstations, while the working modules were placed on the Sun execution nodes. This experiment also demonstrates execution using heterogeneous platforms. The results are summarized in Figure 5.

		Run 1	Run 2	Run 3	Average
MARS	real	23.65	24.06	23.12	23.61
ORIGINAL	real	69.4	69.0	69.5	69.30

Figure 5. Partial 2-way pipeline results

Figure 5 shows the results of adding two more modules to the collection. The original setup now takes 194% longer than the MARS module collection, or about three times as long. The new setup is faster than the simple pipeline, 23.61s versus 35.67s, a savings of about 12.06s, or faster by just more than one-third of the simple pipeline's total time. Finally, the two-stage pipeline performance about is three times better than the original setup.

This stage-depth parallelism can be arbitrarily extended at any point. We extend the example by adding another *is* and another *rn* module. The leader code requires new information to *locate* the new modules on the network, but really nothing more. The control loop in the leader is changed by only *one character*. In the loop, the fragment `if (done % 2 == 0)` is changed to `if (done % 3 == 0)` to indicate that there are three copies of the parallel modules, rather than two. That is the only change. The resulting configuration is shown in Figure 6.

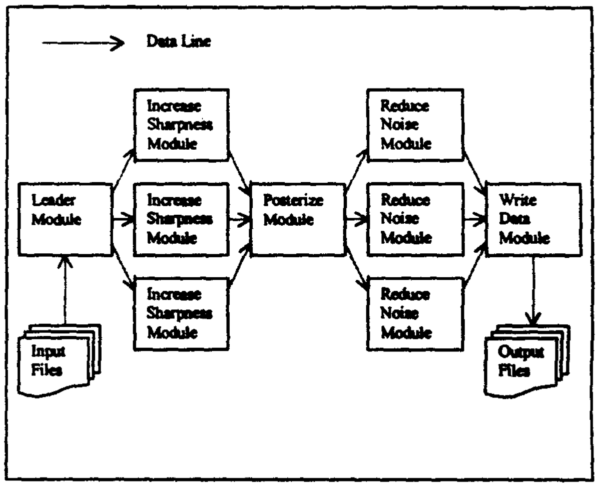

Figure 6. Partial 3-way pipeline

As expected, the execution time is again decreased. The results from the partial three-way pipeline are shown in Figure 7.

		Run 1	Run 2	Run 3	Average
MARS	real	19.02	18.85	19.03	18.97
ORIGINAL	real	69.4	69.0	69.5	69.30

Figure 7. Partial 3-way pipeline results

As before, the runtime is reduced. The amount of time reduction by adding this extra set of modules is not as large as the reduction from the first extra set added. (The original code still takes *265% longer* than the MARS code) A single leader reading inputs cannot keep the pipeline completely filled, nor can a single data writing module commit the data to disk fast enough to scale further, without adding copies of *po* and the write-out module. Three *is* and *rn* stages are all this pipeline can support. Of course, if there were more data readers and writers, and if enough execution nodes were available, the problem could be easily scaled to any arbitrary level, with minimal changes to the leader code. This flexibility allows various scalable problems to be mapped onto numerous configurations of processors with minimal changes to module codes.

This example has demonstrated several more elements of the MARS design. First, MARS has been built to make scaling problems simple. Once a problem has been broken into its parallel components, arbitrary scaling at each stage of the pipeline is a relatively easy process. Also, the partial two- and three-stage pipelines show the support for heterogeneity in the MARS system. The modules in those examples were executed on different platforms, while data and control messages required no alteration. Since the data in the examples was binary, no conversion between platforms was required either.

Biology Model

Description

The MARS system test would not be complete without applying it to an actual scientific MDO code. Each of the codes tied together by hand is unique, and handwritten codes using similar modules vary widely in implementation method and performance. The system we chose to test MARS has three primary modules, including a handwritten leader-like code [6].

A complete description of the code we applied the MARS system to can be found in "Global net carbon exchange and intra-annual atmospheric CO_2 concentrations predicted by an ecosystem process model and three-dimensional atmospheric transport model" [6].

In the original system, the leader was known as "gessys". The gessys module served the same role as a leader module in a MARS ensemble. The other two working modules were labeled climnew and globebgc. Some of the data transfers and module invocations performed by the gessys code were achieved with the following fragment:

```
system("./climnew");
system("./globebgc");
system("cat grid.day >> newout.day");
system("cat grid.grw >> newout.grw");
system("rm -f grid.mtc");
system("rm -f grid.clm");
```

The three modules were wrapped and tied together into a MARS ensemble. The ensemble is shown in Figure 8. The disk-based output was left in the globebgc module. The data files generated for a single run were very large (over seventy megabytes) and took a significant amount of the total runtime to write out. The output was left on secondary (disk) storage in the MARS implementation because it was also available after the original system ran.

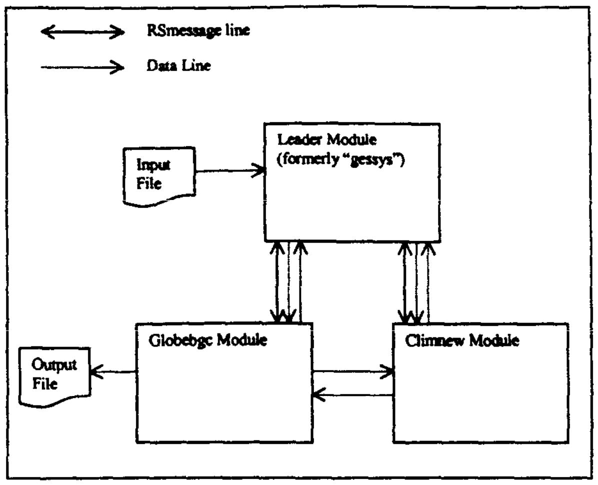

Figure 8. Biology module arrangement

Results

The results shown in Figure 9 shows the performance on the SGIs with a small data set. The original setup takes 123% longer to run on the SGIs than the MARS version of the biology code. This is an excellent time improvement; while only one extra CPU is being used, there is a speedup of over 100%. The original *problem* scales perfectly (1:1) and still goes a bit further. The original *code* could not be readily scaled across two CPUs (even with identical file systems) because there are no synchronization guarantees with UNIX files. One module had to run to completion before the second was executed. That problem was solved with our binary I/O implementation.

		Run 1	Run 2	Run 3	Average
MARS	real	17.49	17.49	17.67	17.55
ORIGINAL	real	39.38	39.07	39.22	39.22

Figure 9. Small data set on the SGIs

CONCLUSIONS

MARS proved to be a successful solution to the problem of integrating multidisciplinary codes. There were many features of MARS that contributed to its overall success. Those primary achievements are outlined here.

Improved times

On each the test cases, there were significant speedups. Overall runtimes were improved in each of the three main examples (pipelined graphics filtering, extended pipelined graphics filtering, and the biology code).

In each case, performance of the MARS ensembles was very good. The extreme performance gain was shown in Figure 7, the partial three-way graphics pipeline results. The original code took 265% longer than the MARS modules or about 3.65 times the total MARS time. Were we to discount setup overhead and assume already running modules, that time would also be further improved. Also, we timed the ensemble with synchronous module terminations, which can be significant with the nine modules involved (Figure 6). Using an asynchronous termination call could have yielded better results as well.

Overall system load is reduced by using MARS data paths (where possible) instead of file based I/O. This is an important consideration. If one machine can be used more efficiently, it frees up valuable cycle time for other processes and/or additional MARS modules. For instance, in our three-way pipeline parallelism example, we placed several modules on a single-CPU SGI and saw excellent performance because significantly less time was spent reading, writing, and verifying data previously written to disk. This time can be used for additional MARS modules or for any other processes the user chooses.

Distribution and realignment

There are four distribution and realignment advantages of the MARS system. The first is that leader modules can be changed slightly and effect significant changes in the overall setup of working modules. The second advantage is that altering parallel codes is a simple process. The depth of particular stages of the ensemble can be changed from the leader code without altering the working modules. The third advantage leverages off the first two: at runtime, sophisticated leaders can add and remove modules from running ensembles. The fourth advantage is that the late binding time for module identifications means that leaders can be ignorant of even their working modules until runtime, and the working modules can even be specified from the command line by users.

Heterogeneity

Because data and control messages are passed using an abstract type, MARS implementations can run on multiple platforms without requiring platform-specific messaging code. Of course, platform specific *optimizations* are possible, but do not affect compatibility. The RSMessage type is composed of byte (`char`) fields that are dynamically allocated and word-aligned based on the data types they represent. As such, network transmission and message translation are simple processes.

Since most other MARS functions are based on standard protocols, there has been no problem porting the MARS runtime system to many different platforms. The current runtime system compiles on Sparc, Intel, and MIPS CPUs running Solaris, Irix, and Linux *without any conditional compilation statements*. MARS modules concurrently running on different combinations of OSes and platforms readily cooperate and exchange data and control messages without difficulty.

Conditional execution

By augmenting an ensemble of working modules with *testing* modules, the ensemble can be terminated early or runtime can be extended as needed. In addition, certain modules can be executed (or *not* executed) as needed, based on the outputs of testing modules. For instance, imagine an iterative approximation process (like Newton's method) designed to calculate a numerical result. With poor starting values, such processes can quickly diverge. It is of great benefit to be able to detect such divergence quickly and restart the process with different initial values. By the same token, a converging process that has not yet achieved a certain level of precision should be extended rather than terminated and subsequently restarted with an increased number of iterations.

SUMMARY

The primary goal of the MARS project is to build a runtime system to support multidisciplinary applications. These applications are formed from collections of modules that often include legacy codes, not just newly created modules. While MARS supports legacy codes, it is still an excellent authoring system for distributed codes. The working modules that form an ensemble cooperate in a heterogeneous execution environment, frequently composed of clusters of workstations.

MARS modules can be invoked as part of an ensemble or alone. As such, MARS supports *single-source* coding, making code maintenance easier for ensemble creators.

The runtime system supports *dynamic invocation*, *termination*, and *realignment* of modules. Module distribution is a simple task; it does not require a registration step like CORBA and other invocation strategies. While dynamic module realignment is a simple task, dynamic *redirection of the user data* passed between modules is possible as well. Primitives for *conditional execution* also exist in MARS.

Even with all these control considerations, *performance* is still central to MARS' implementation. MARS does not add significant overhead for its control or data communications to multidisciplinary applications. MARS can be used to distribute and parallelize collections of otherwise stand-alone modules into cooperative ensembles.

MARS is designed to take expert knowledge embodied in multiple computer programs and glue those programs together. The program ensembles formed by this process are used to solve problems larger than any component module can solve.

FUTURE WORK

While designing and working with the MARS system, several additional components were suggested. Some of these modifications have clear implementation paths, while others are still open ended problems. With these proposed features in place, MARS would be one of the most fully functional distributed runtime support system available. With the support for integrating existing multidisciplinary codes, these enhancements effectively remove any limits to designing new applications or modifying existing ones.

Enhanced dynamic scheduling, multi-language support, native threading, and automatic data conversion are projects currently being examined.

REFERENCES

[1] R. Adler, "Distributed Coordination Models for Client/Server Computing," Computer, April 1995.

[2] H. Bal and M. Haines, "Approaches for Integrating Task and Data Parallelism," Technical Report IR-415, Vrije Universiteit, Amsterdam, December 1996.

[3] B. Chapman, M. Haines, P. Mehrotra, H. Zima and J. Van Rosendale, "Opus: A Coordination Language for Multidisciplinary Applications," ICASE Technical Report 97-30, June 1997.

[4] M. Haines, B. Hess, P. Mehrotra, J. Van Rosendale and H. Zima, "Runtime Support For Data Parallel Tasks," *Proceedings of The Fifth Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, February 1995.

[5] M. Haines and P. Mehrotra, "Exploiting Parallelism in Multidisciplinary Applications Using Opus," *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, San Francisco, CA, February 1995.

[6] E. Hunt, S. Piper, R. Nemani, C. Keeling, R. Otto and S. Running, "Global Net Carbon Exchange and Intra-Annual Atmospheric CO_2 Concentrations Predicted by an Ecosystem Process Model and Three-Dimensional Atmospheric Transport Model," Global Biogeochemical Cycles, September 1996.

[7] P. Mehorotra and M. Haines, "An Overview Of The Opus Language And Runtime System," ICASE Technical Report 94-39, May 1994. (Also appears in *Languages and Compilers for Parallel Computers*, Springer-Verlag Lecture Notes in Computer Science, vol 892, Pages 346-360.)

[8] L. Perrochon, G. Wiederhold and Burback, "A Compiler for Composition: CHAIMS," *Fifth International Symposium on Assessment of Software Tools and Technologies* (SAST'97), Pittsburgh, PA, June 3-5 1997.

[9] C. Tornabene, P. Jain and G. Wiederhold, "Software for Composition: CHAIMS," *Workshop on Compositional Software Architectures of OMG, DARPA and MCC*, Monterey, CA, January 6-8 1998.

[10] C. Tornabene, D. Beringer, P. Jain and G. Wiederhold, "Composition on a Higher Level: CHAIMS," *unpublished*, Dept. of Computer Science, Stanford University.

[11] G. Wiederhold, P. Wegner and S. Ceri, "Towards Megaprogramming," Communications of the ACM, November 1992.