Marionette: a System for Parallel Distributed Programming Using a Master/Slave Model

Mark Sullivan
David Anderson

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, California 94720

November 8, 1988

ABSTRACT

Marionette is a system for distributed parallel programming in an environment of networked heterogeneous computer systems. It is based on a master/slave model. The master process can invoke *worker operations* (asynchronous remote procedure calls to single slaves) and *context operations* (updates to the state of all slaves). The master and slaves also interact through *shared data structures* that can be modified only by the master. The master and slave processes are programmed in a sequential language.

The Marionette runtime system manages slave process creation, propagates shared data structures to slaves as needed, queues and dispatches worker and context operations, and manages recovery from slave processor failures. The Marionette system also includes tools for automated compilation of program binaries for multiple architectures, and for distributing binaries to remote file systems. A UNIX-based implementation of Marionette is described.

This work supported by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871, monitored by Naval Electronic Systems Command under Contract No. N00039-84-C-0089, and by a California Microelectronics Fellowship.

1. INTRODUCTION

The computing facility of the UC Berkeley CS Division includes numerous diskless workstations, some file servers, and a few mainframes with local file systems. These systems are interconnected by LAN's, and all run variants of BSD UNIX. At any given time, most of the processors are idle.

Marionette is a system for parallel distributed programming. It is intended to give average programmers convenient access to the potential parallel processing power of environments like that of the CS division. Its programming model deviates little from sequential models, yet allows significant parallelism for a large class of applications. It is easily portable to standard UNIX systems.

In the Marionette programming model there is one *master process* and many *slave processes*. These processes are sequential, and they interact in several ways (see Figure 1):

- The master may create and update *shared data structures*. Slaves can read, but not write, copies of these structures (because of processor heterogeneity, the slave copies may have different data representations).
- The master may invoke *context operations*, which are executed on all slaves and may modify the slave process state. They have arguments but do not return results.
- The master may invoke *worker operations*. Each is executed on a single slave, and returns a value; however, the master does not specify which slave is used. Worker operations are invoked asynchronously, and the master must later explicitly *accept* the operation's results. By invoking several worker operations before accepting their results, an application can achieve parallel distributed execution. If several worker operations are invoked

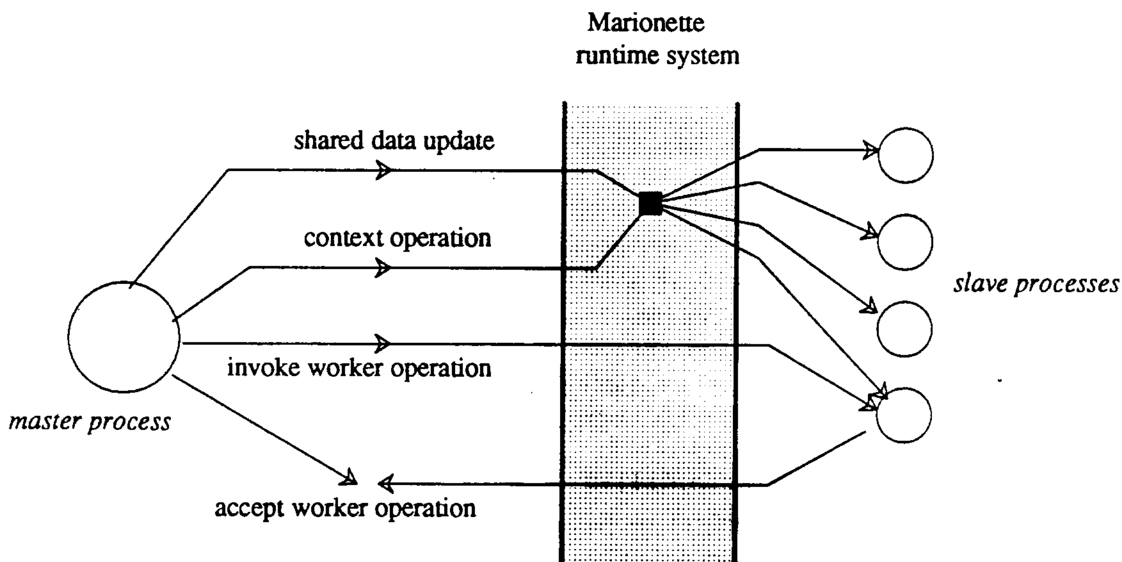

Figure 1: The Marionette Programming Model

concurrently, they do not necessarily return in the order in which they were invoked.

The Marionette runtime system (currently implemented as a pair of user-level libraries) has several functions:

- It manages slave process creation. The set of slave hosts is specified in a configuration file. If a processor becomes available (or recovers from a crash) during program execution, it is added to the slave pool.
- It queues worker and context operations, and dispatches them to a set of slave processors.
- It detects and manages recovery from hardware or software failures in slaves. A worker operation in progress on a crashed slave is restarted on another slave.
- It propagates shared data structures to slaves as needed, and converts them to the appropriate data representation.

As an example use of the Marionette programming model, consider the task of generating an animated film using ray-tracing. To parallelize this task using Marionette, the master program might be structured as follows:

(1) Create a shared data structure describing the initial 3-D scene.

(2) Invoke a set of worker operations to compute a frame. Each operation computes one or more pixel values by "tracing rays" through the 3-D scene.

(3) Modify the shared data structure to define the scene in the next frame.

(4) Accept the results of the current round of worker operations, and write the image to videotape or disk.

(5) If more frames remain, go to (2).

The execution of this program will have two important properties. First, the pixel-value computations will be dynamically distributed among the set of available processors and executed in parallel. Second, the (presumably large) data structure defining the 3-D scene is copied to the slaves in its entirety only at the start of the program. Thereafter, only the portions changed between frames are copied.[1] The details of both these mechanisms are hidden from the programmer.

Marionette has been implemented in C on UNIX, and runs on VAX and Sun computers. We have used it to write several applications, including a parallel version of an existing ray-tracing renderer [15] that follows the above structure. The implementation includes *mdist*, a tool for automated compilation and distribution of program binaries for multiple architectures.

This paper is organized as follows: Sections 2 and 3 describe the abstractions and implementation of the Marionette runtime system. Section 4 describes the *mdist* utility. Section 5 discusses Marionette's design decisions and compares it to related systems. Section 6 gives our conclusions. A complete Marionette application program is shown in an Appendix.

2. MARIONETTE PROGRAMMING PRIMITIVES

The semantics of Marionette's primitives are based on the notion of *slave state*. The state of a slave process is determined by the sequence of context operations and shared data updates it has received. At any point in the execution of the master process, the sequence of context operations and shared data updates up to that point defines the *logical* slave state.

[1] Instead of storing the scene description as a shared data structure, we could store it as a local data structure on each slave and use context operations to distributed and update it. This might be preferable if the updates change large amounts of data but can be described by small messages.

Actual slave states are maintained using a "lazy update" policy. The result of a worker operation is a function of 1) its arguments and 2) the logical slave state when the operation is invoked. Before a worker operation request is sent to a slave, the minimal set of data structure updates and context operations needed to establish the correct slave state is sent first. Therefore, the states of slaves may lag behind the logical slave state.

2.1. Function Identifiers

Since the master and slave programs may run on different machine types, all data passed between them is put into a common representation. We use Sun's External Data Representation (XDR) [14] for this purpose. The programmer must supply XDR routines for all data types passed between master and slave. These routines convert data between a machine-dependent form and a serialized, machine-independent form.

The master and slave programs are independently compiled, and need a common way of referring to functions. This is done by requiring all programs to define the following *linkage arrays*; a function identifier is an index into one of the arrays.

- The array `worker_ops` contains descriptors for worker operation functions. Each descriptor contains pointers to the function (defined only in slaves), to an XDR routine for argument conversion, and to an XDR routine for result conversion.
- The array `context_ops` contains descriptors for context operation functions. Each descriptor contains pointers to the function and to an XDR routine for argument conversion.
- The array `shared_data_types` contains descriptors for types of shared data structures. Each descriptor points to an XDR routine for converting that type.

2.2. Shared Data Structures

Marionette allows data structures to be "shared" between master and slaves. These *shared data structures* may be linked structures such as lists or graphs. They may be created, augmented and modified during execution. Each shared data structure must have a *root*; for example, for a linked list this would be the first node in the list. For a shared array, the root is the entire structure. The XDR routine for a shared data type (see above) must traverse and serialize the entire structure, given a pointer to the root.

After the master has allocated and initialized a shared data structure, it must *register* it with the runtime system before the structure can be passed to slaves. This is done using:

```
register_shared_structure(
        void* root,         // pointer to the root of the data structure
        int   type          // index into shared_data_types
);
```

The address and size of the root may not be changed after the structure is registered. If the root is dynamically allocated, it must not be freed and reused for a different purpose.

If the master modifies a shared data structure, it must notify the runtime system before invoking further context or worker operations. This is done using:

```
update_shared_structure(
        void* root          // pointer to the root of the data structure
);
```

Marionette provides an XDR primitive `xdr_shared_struct_ptr()` to (de)serialize pointers to shared data structures. Arguments to worker and context operations may include pointers to shared data structures. For example, a context operation can install a pointer to a shared data structure in a global variable of the slave program (this is, in fact, the only way that such a variable can be initialized). Slave processes are normally not allowed to modify their

copies of shared data structures[2].

The choice of "granularity" in shared data structures can affect performance. if any part of a shared data structure is modified, the entire structure is copied to slave processes. Hence in some cases it might be preferable to register each node of a tree (rather than the entire tree) as a separate shared data structure.

2.3. Worker Operations

The master invokes and accepts worker operations using

```
int                                 // returns QUEUE_FULL flags (see below)
invoke_worker(
        int         function,       // index into worker_ops array
        int         instance,       // caller-supplied UID for this operation
        void*       args,           // arguments to operation
        void*       results,        // results of operation
        BOOLEAN     asynch_copy     // whether to copy results asynchronously
);

BOOLEAN                             // returns false if no outstanding operations
accept_worker(
        int         *instance       // instance ID (returned)
);
```

`Invoke_worker()` asynchronously invokes a worker operation, and `accept_worker()` waits for a worker operation to complete. The arguments to `invoke_worker()` are as follows.

`Function` is the function identifier for the worker operation.

`Instance` is supplied by the master. When an operation finishes, its instance ID is returned by `accept_worker()`.

`Args` points to the worker operation's arguments. They are copied (by the XDR routine in `worker_ops`) before `invoke_worker()` returns, so they can be modified immediately thereafter.

`Results` points to a buffer in which the results are to be stored.

If `asynch_copy` is true, the results are copied into `results` as soon as possible (potentially before the master calls `accept_worker()`). This potentially increases performance, but requires that the master allocate separate return buffers for each outstanding operation. If `asynch_copy` is false, copying is done during the call to `accept_worker()` that accepts this particular worker operation. The master can then use a single result buffer.

The Marionette runtime system maintains two queues in the master process: the *pending queue* stores pending worker operations, and the *done queue* stores the results of worker operations that have been completed but not accepted. The size of these queues is bounded.

`Invoke_worker()` is non-blocking. If neither queue is full it dispatches or queues the operation and returns zero. Otherwise, the worker operation is not dispatched, and two flags `DONE_QUEUE_FULL` and `PENDING_QUEUE_FULL` are returned. If `DONE_QUEUE_FULL` is set, the master must accept some worker operations before it can invoke more. If `PENDING_QUEUE_FULL` is true, it may either accept worker operations, or it may call

[2] This rule can be relaxed if slave copies are modified only by context operations. This may improve performance when updates can be described by small messages, but involve changes to large amounts of data.

```
pending_queue_wait();
```

which blocks until the pending queue is not full.

In general, the master should try to keep the slaves as busy as possible. Assuming that there are no dependencies between worker operations, this can be done as follow:

```
while (more work to do) {
      if (invoke_worker(func, inst, in, out, FALSE) != SUCCESS) {
            accept_worker(&inst);
      }
}

while (accept_worker(&inst));
```

In the first loop, worker operations are generated as fast as flow control permits. The second loop accepts any remaining results. Variations of this structure can be used in cases where synchronization points are required because of dependencies between worker operations. For example, suppose that in the ray-tracing program in Section 1 there only enough buffering for a single frame. The master must then assemble an entire frame before issuing the worker operations for the next frame.

2.4. Context Operations

Context operations are used to modify slave state. A context operation might allocate and initialize a large data structure, open and read an initialization file, or store the address of a shared data structure in a static variable. The master invokes context operations using

```
STATUS
invoke_context(
      int          op_id,             // index into context_ops table
      void*        arguments,
```

`Op_id` is a function identifier for the operation. Slaves execute the context operation with the given `arguments`.

As mentioned above, `invoke_context()` does not immediately dispatch the operation to all slaves. It merely enqueues the operation and dispatches it on demand as needed for subsequent worker operations.

3. MARIONETTE IMPLEMENTATION

The Marionette runtime system is implemented by a *master library* linked with the master program, and a *slave library* linked with the slave program (all slave processes normally execute the same source program, perhaps compiled for different machine architectures).

3.1. Process and Communication Structure

All slave hosts must run an application-independent *daemon process*, perhaps created at boot time. Execution of a Marionette program begins with a *communication setup* phase. During this phase, the master process contacts the daemon process at each potential slave host and has it create a slave process executing the application's slave program binary. The slave process communicates with the master over a TCP connection [18].[3]

---

[3] Any reliable request/reply facility could be used instead. TCP connections are handy because they provide failure detection. However, each TCP connection requires a file descriptor in the master, imposing a limit (currently 64) on the number of slaves.

After the communication setup phase, the master library calls `master_main()` to execute the application program. Using the UNIX timer mechanism, the library periodically checks for slave hosts that have just recovered from a crash. Messages from the slaves are handled asynchronously by signal handlers [4]. The library uses signal-masking to serialize its access to internal data structures.

3.2. Master Library Data Structures

The master maintains the following data structures (see Figure 2):

- *Slave state history*: a time-ordered list of all context operations and shared data structure updates. Each list entry contains the network message to be sent. Entries for shared data updates include a pointer to the previous and next update records for the structure, if any.
- *Slave descriptors*: the descriptor for an active slave contains a pointer into the slave history list indicating the current state of the slave (i.e., the operation sent to it most recently). The descriptor also contains pointers to the descriptors (see below) for worker operations currently in progress on the slave.
- *Outstanding worker operations*: descriptors for worker operations invoked but not yet completed. The descriptor includes a pointer into the slave history list for the state in which the operation is to be executed. If the operation has been dispatched to a slave, the descriptor includes a pointer to the slave descriptor.
- *Completed worker operations*: descriptors for worker operations that have been completed but not yet accepted.

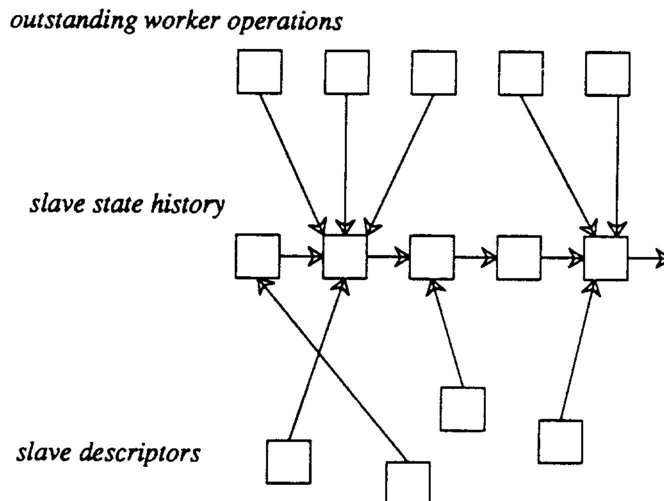

Figure 2: Slave-State Data Structures in the Master Process.

3.3. Scheduling Operations

When a worker operation is invoked, the runtime system prepares a request message, calling the XDR routine to serialize the operation's arguments. If there is a slave that is not currently executing a worker operation, the request message is sent to that slave. Otherwise, the request is enqueued and sent when a slave is free.

Before sending a request message to a slave, the runtime system must establish the operation's slave state S on that slave. It does this be scanning forward in the slave state history from the slave's current state up to S. Context operations are dispatched to the slave as they are encountered. A shared data update U is dispatched only if there is no update to the same structure intermediate between U and S.

3.4. References to Shared Data Structures

The unique ID of a shared data structure is the address of its root block on the master. The slave library maintains hash tables mapping these addresses to and from local addresses. The XDR routine `xdr_shared_struct_ptr()` is implemented as follows. To serialize on the master, the master address is written (the pointer is not followed). To deserialize on the slave, the master address is converted to a local address using the translation table. To serialize on the slave (e.g., if a pointer to a shared data structure is passed in a return value) the local address is converted to the master address.

3.5. Recovery from Slave Failures

A slave *fails* when either its process crashes, its host crashes, or the network connection is broken. These conditions all result in the failure of the TCP connection, which is easily detected by the master. Any outstanding worker operations are rescheduled on one of the surviving slaves. After a failure, the master periodically attempts to restart the slave.

When a slave fails, it may be executing a worker operation. The runtime system will reschedule this operation on another slave. If the slave state of the failed operation is greater than or equal to that of some running slave, the operation can be directly scheduled on that slave. Worker operations have no side effects, so they can be restarted many times.

If the slave state S of a failed worker operation precedes that of any other slave, then a slave must be "rolled back" to accommodate the operation. To restart the worker operation, the master selects an idle slave, waiting if necessary. Let T denote the slave's current state. If the slave state history between S and T includes only shared data updates, then these updates are undone (each data structure involved is restored to its state as of S). If, however, the state history between S and T includes context operations, then the state history is "replayed" from the beginning up to S.

4. COMPILING, EXECUTING AND DEBUGGING MARIONETTE PROGRAMS

In our UNIX-based implementation, a Marionette program has a *home directory* on which its sources are kept. For simplicity, this is on the file system of the master. The user of a Marionette program must provide two *configuration files* in the home directory:

- An *application description file* giving application-dependent information such as the names of the source and input files.
- A *host configuration file* describing the set of slave hosts. These are divided into *groups*; hosts in a given group have the same processor and operating system type and can therefore use the same slave binary. The configuration file also specifies which hosts are clients of a common file server, and can therefore share a single copy of the binary.

The programmer maintains source files only in the home directory. The Marionette system includes a program *mdist* for compiling and distributing binaries and input files. *Mdist* uses the UNIX utilities *make* [7] and *rcp* [4]) to distribute the slave program source, to compile the sources (only one compilation per slave group is needed). After compilation is completed, the slave binaries are copied to other file systems as needed. In addition, *mdist* concurrently compiles the master program.

When *mdist* is finished, the user simply runs the master program. The master uses the host configuration files to create slave processes as described in Section 3.1, and then transfers control to `master_main()`. When this returns, the runtime system kills all slave processes.

An example is shown in Figure 3; *FS0* and *FS1* are file servers for groups of workstations to be used as slaves. They and their clients all have the same machine architecture and therefore can use the same slave binary. In this case, *mdist* does the following:

(1) *VAX0* sends slave sources to *FS0* and *VAX1*.

(2) *FS0* and *VAX1* compile the slave binary.

(3) *FS0* sends its binary to *FS1*; *VAX1* sends its binary to *VAX2*.

On completion, the user runs the master program on *VAX0*, which starts slave processes on both Vax's and all the workstations.

4.1. Debugging Marionette Programs

The Marionette system includes a "single-process" version of the runtime system. This library is linked with both the master and slave programs, producing a single "prototype" program. The developer can use a traditional single-process debugger (e.g., *dbx* [4]) to debug the

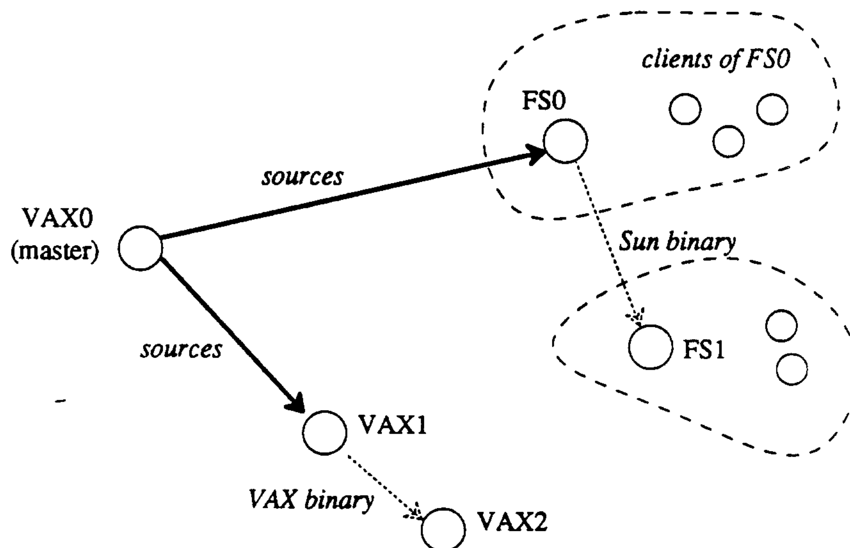

Figure 3: Compilation and Distribution using *mdist*.

prototype program. Since the two versions of the Marionette library provide the same interface, the program can then be changed from single-process to distributed form with no source modifications.

The single-process library does the following:

- `Invoke_worker()` simply calls the operation function directly. The XDR routines are called to serialize and deserialize the arguments.
- `Invoke_context()` calls the operation function directly.
- The user can select whether there is a single copy of shared data structures (in which case `Xdr_shared_data_ptr()` is a no-op) or two copies of shared data structures are maintained. The latter choice is slower but can detect errors in which update operations are mistakenly omitted in the master.

5. DISCUSSION AND COMPARISON

This section discusses the tradeoffs in each part of the Marionette design, and compares Marionette with related systems.

5.1. The Master/Slave Model

The salient aspect of the Marionette design is its master/slave model. Slaves can execute worker and context operations, but cannot communicate with one another directly. Furthermore, the state of a slave is determined by the operations sent to it, and is therefore known to the master at all times. This organization has the following advantages:

- *Simple recovery from slave crashes.* Since worker operations do not communicate with one another, the only part of the computation affected by a slave failure is the worker operation that is executing on the failed slave. Restarting that worker operation will reproduce the lost work without changing the ultimate outcome of the computation. If direct communication between all processes were allowed, recovery from failures would require a more complex mechanism such as distributed transactions [9, 17].
- *Configuration independence.* Marionette separates *logical* parallelism (expressed in the an application program) from *physical* parallelism (managed by the runtime system, and limited by the processor configuration). Changes in the number or types of processors available do not affect the program or its results.
- *Load-balancing.* Worker operation computation is automatically load-balanced; fast or lightly-loaded processors finish their operations first, and are issued more operations.
- *Freedom from deadlock.* Worker operations never block. The master blocks only to wait for a worker operation to complete. The program may cease to make progress if all slaves fail, but cannot deadlock.

The master/slave model has three potential disadvantages:

- The master is a single point of failure. However, a conventional non-distributed program also has a single point of failure, and a Marionette program for the same task is potentially more fault-tolerant simply because it usually will finish faster.
- The master process could be a performance bottleneck, limiting the effective parallelism. Assuming that the program can be structured to produce unlimited concurrent worker operations, performance depends on two factors: the rate at which the master can dispatch and accept worker operations, and the time needed by slaves to execute worker operations. In our Marionette implementation running on a Sun 3/50, it takes about 12 milliseconds of CPU time for the master to dispatch and accept a worker operation with a one-word argument and result. Hence if these worker operations (including serialization and

communication overhead) consume 1 CPU second each, about 80 slave processors could be kept busy. At another extreme, suppose a worker operation can be dispatched and accepted in 2 milliseconds (a typical figure in optimized RPC systems) and each operation uses 100 CPU seconds. In this case about 50,000 slaves can be effectively used.

- Some tasks may be difficult to express in the master/slave model. This may be the case when the natural communication structure of the problem follows a problem-dependent graph structure (shortest-path problems, tree search, etc.).

In contrast with the master/slave model, many parallel languages are based on a model in which processes have a peer relationship, and can communicate with arbitrary other processes. This communication may be based on message passing [3, 10] (of which Remote Procedure Call (RPC) [1] is a special case) or simulated shared memory [8, 11, 12]. By placing fewer restrictions on interprocess communication, this model has greater flexibility than Marionette. However, it loses the advantages of simple recovery and freedom from deadlock. Furthermore, dynamic load-balancing can be accomplished in this model only with process migration, which can be complex and expensive.

Researchers have implemented forms of RPC in which it is possible to execute multiple concurrent operations. The systems described by Cooper [5] and Martin [16] offer constructs that invokes many parallel operations at once and then block. Both of these systems are intended for fault-tolerance rather than for parallel programming. A single invocation causes several instances of the same remote operation to execute on different hosts in parallel. It would be possible to offer a primitive initiating many different remote operations; the caller would block until all the operations had completed. Compared to Marionette, this would limit parallelism without any clear conceptual advantage.

The Mercury system [13] has a parallel RPC mechanism intended for pipelining operations to a single server. There are several differences between a sequence of parallel operations in Mercury and a sequence of Marionette worker operations. Marionette dispatches operations to a *pool* of slave processes, while Mercury sends operations to a single remote process. Mercury calls execute and return in the order in which they were invoked.

5.2. The Invoke/Accept Model

In Marionette, the master process generates concurrent operations by asynchronous invocation (using invoke/accept). As far as the programmer is concerned, the master process is never interrupted. It only receives replies from slaves when it explicitly accepts them. The master must keep track of how many worker operations, and perhaps which operations, are outstanding.

A small but important aspect of the Marionette design is that the master assigns the "instance ID's" of worker operations. These ID's can encode the purpose of the operation. In the ray-tracing example, the ID might encode the coordinates of the pixels involved. This eliminates the need to maintain a mapping from system-supplied ID's to per-operation information, which is sometimes cited as a disadvantage of asynchronous operations [2].

An alternative organization for the master program is as multiple "lightweight" processes sharing an address space. Distributed parallelism can then be achieved by having several processes make (synchronous) remote requests. The processes must synchronize using monitors, fork/join, etc. Stumm [20] and Crow [6] describe parallel applications written in this style on the V system and Cedar, respectively.

Relative to lightweight processes, the Marionette approach has several advantages. There is no concurrency in the master's address space, so there is no need for mutual exclusion in the applica-

tion program [4]. There is no context switch or process creation overhead. Finally, because no lightweight process mechanism is needed, the implementation is simpler and more portable.

5.3. Shared Data Structures

Shared data structures were included in the Marionette design to improve communication efficiency. In theory, a worker operation's arguments could include all the data it needs. Every worker's arguments, however, are copied to the slave host when the operation is executed. The shared data structure facility allows a program to tell the runtime system that certain data structures are used by many workers and seldom modified. The runtime system can then cache these data structures at the slave rather than recopy them during every worker operation.

Marionette's shared data structure facility is related to network shared memory [12, 21]. Our approach has has several advantages.

- Marionette allows sharing between heterogeneous machines.
- Since Marionette allows slave state inconsistency, parallel computation can take place on slaves that have different versions of a shared data structure.
- Marionette runs on any UNIX system supporting the Internet protocols; no special VM support is needed.

5.4. Context Operations

Context operations allow an application to propagate large data structures in a compressed form. For example, in the ray-tracing program context operations could potentially be used in the following ways:

- A context operation could distribute the initial 3-D scene expressed in a ''scene description language'' such as Unigrafix [19] (this language is already used to reduce the size of scene description disk files). The context operation expands it into a large data structure.
- A context operation could cause each slave to read the scene description from a local disk file.
- Context operations could be used to efficiently propagate changes to scenes that can be computed quickly but which modify large amounts of data (e.g., moving many points by a fixed amount).

6. CONCLUSION

Marionette is a system for parallel distributed programming for a network of heterogeneous hosts. Marionette uses a restricted *master/slave* process model in which worker operations are invoked asynchronously. *Shared data structure* and *context operation* mechanisms allow efficient read-only sharing of data structures by slaves.

The Marionette design allows large-scale parallelism for a variety of applications. In addition, it permits simple recovery from slave failure, requires no complex synchronization mechanism, and can be implemented as a library for a sequential language.

The Marionette system also includes tools for automatic distribution and compilation on heterogeneous systems, and provides a single-process version of the runtime system to facilitate debugging using existing tools.

---

[4] Except for the optional copying of worker operation results into user buffers. This limited form of concurrency is simple to manage.

6.1. Future Work

Having implemented Marionette and used it for a few applications, our first goal is to apply it to more diverse applications. Beyond this, we anticipate several future extensions to Marionette.

- We would like to use shared-memory multiprocessor (SMMP) machines. To use an SMMP as the master, the master program would remain sequential, but other functions (XDR conversion, generation of outgoing operations, and handling of incoming messages) could be handled by separate processes in the same address space. To use an SMMP as a slave, multiple worker operations could be done in parallel in a single address space. Alternatively (but less efficiently) slave processes could be started in different address spaces.
- If slave processors have radically different speeds, a situation could arise in which fast slaves are idle, waiting for operations on slower slaves to finish. To avoid this, the master's scheduling policies may have to be changed. For example, worker operations with longer expected durations might be assigned to faster slaves. If there are more slaves than outstanding operations, a single worker operation could be invoked on several slaves.
- The scheduling policy could be modified to *pipeline* operations (i.e., to maintain several outstanding operation per slave), or to *batch* multiple operations. These techniques could potentially increase performance by reducing communication overhead, or by overlapping communication and computation.
- The single-master performance bottleneck could potentially be alleviated using a tree-structured organization. The master process would issue coarse-grained worker operations to *foreman* processes, which would divide them into fine-grained operations, execute these on slave processes, collate the results, and return this to the master.

REFERENCES

1. A. Birrell and B. Nelson, "Implementing Remote Procedure Calls", *ACM Transactions on Computer Systems 2*, 1 (Feb. 1984), 39-59.
2. A. P. Black, "Supporting Distributed Applications in Eden", *Proc. of the 10th ACM Symp. on Operating System Prin.*, Orcas Island, Eastsound, Washington, Dec. 1-4, 1985, 181-193.
3. D. R. Cheriton, "The V Kernel: a Software Base for Distributed Systems", *IEEE Software 1*, 2 (Apr. 1984), 19-43.
4. *The 4.3BSD UNIX Programmer's Manual, Volume 1*, Computer Science Division, EECS, UCB, Berkeley, CA, 1986.
5. E. Cooper, "Replicated Distributed Programs", *Proc. of the 10th ACM Symp. on Operating System Prin.*, Orcas Island, Eastsound, Washington, Dec. 1-4, 1985, 63-78.
6. F. C. Crow, "Experiences in Distributed Execution: A Report on Work in Progress", *SIGGRAPH*, 1986. Notes from work in progress presentation.
7. S. I. Feldman, "Make - A Program for Maintaining Computer Programs", *Software - Practice and Experience 9*, 4 (Apr. 1979), 256-265.
8. D. Gelernter, "Parallel Programming in Linda", *Proceedings of the International Conference on Parallel Processing*, Aug. 1985, 255-263.
9. R. Haskin, Y. Malachi, W. Sawdon and G. Chan, "Recovery Management in QuickSilver", *Trans. Computer Systems 6*, 1 (Feb. 1988), 82-108.
10. C. A. R. Hoare, "Communicating Sequential Processes", *Comm. of the ACM 21*, 8 (Aug. 1978), 666-677.
11. P. J. Leach, P. H. Levine, B. P. Douros, J. A. Hamilton, D. L. Nelson and B. L. Stumpf, "The Architecture of an Integrated Local Network", *IEEE Journal on Selected Areas in Communication 1*, 5 (Nov. 1983), 842-857.
12. K. Li, *Shared Virtual Memory on Loosely Coupled Multiprocessors*, Ph.D. Dissertation, YALEU/DCR-492, Yale University, Sep. 1986.
13. B. Liskov, T. Bloom, D. Gifford, R. Scheifler and W. Weihl, "Communication in the Mercury System", unpublished paper, April 1987.
14. B. Lyon, "Sun External Data Representation Specification", Technical Report, Sun Microsystems, Inc., 1984.
15. D. Marsh, "UgRay: An Efficient Ray-Tracing Renderer for UniGrafix", Technical Report UCB/Computer Science Dpt. 87/360, University of California, Berkeley, May 1987.
16. B. Martin, "Parallel Remote Procedure Call", *CS-097*, 1987.
17. J. E. B. Moss, "Nested Transactions and Reliable Distributed Computing", *Proc. 2nd IEEE Symp. on Reliability in Distributed Software and Database Systems*, July 1982, 33-39.
18. J. Postel, "Transmission Control Protocol", *DARPA Internet RFC 793*, Sep. 1981.
19. C. Sequin, "The Berkeley Unigrafix Tools, Version 2.5", Technical Report 86/281, University of California, Berkeley, December 1985.
20. M. Stumm, "Strategies for Decentralized Resource Management", *SIGCOMM87*, , 245-253.
21. M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black and R. Baron, "The Duality of Memory and Communication in the Implementation of a

Multiprocessor Operating System", *Proc. of the 11th ACM Symp. on Operating System Prin.*, Austin, Texas, Nov. 8-11, 1987, 63-76.

APPENDIX -- AN EXAMPLE MARIONETTE PROGRAM

```
/*
 * This simple Marionette program computes the sum of the squares
 * of the integers from 1 to 100.
 * The squaring is done by slave processes in worker operations.
 * This program can be used for either master or slave.
 */

#include "marionette.h"
#include "xdr.h"

WORKER_OP worker_ops[] = {
        {&squarer, xdr_long, xdr_long}};

master_main()
{
        int i, sum = 0, result, j;

/* invoke worker operations; if queue full, accept one instead */

        for (i=0; i<100; i++) {
                if (worker_invoke(0, 0, &i, &result, FALSE) != SUCCESS) {
                        worker_accept(&j);      /* j (instance ID) is not used */
                        sum += result;
                }
        }

/* accept any outstanding worker operations *

        while (worker_accept(&j)) {
                sum += result;
        }

        printf("the answer is %d0, sum);
}

int*
squarer(i)          /* the worker operation routine */
        int i;
{
        int j = i*i;

        return(&j);
}
```