Parallel and Distributed Computing Solutions

Virtual Shared Memory and the Paradise® System for Distributed Computing

A Technical White Paper
from **SCIENTIFIC**
April 20, 1999

SCIENTIFIC Computing Associates, Inc.
One Century Tower
265 Church Street
New Haven, Connecticut 06510

Tel: (203) 777-7442
Fax: (203) 776-4074
Email: sales@sca.com
Web: www.sca.com

Copyright © 1999 Scientific Computing Associates, Inc. All rights reserved.

A Distributed Computing Example

Suppose your task is to provide a pricing system for the trading and analysis desks of a growing Wall Street investment firm. Your users—investment traders and analysts at several offices worldwide—want to perform pricing and analysis of complex financial instruments, such as fixed income derivative products involving American-style swaptions and index amortized swaps. To satisfy them, you'll need to provide a large amount of computing power.

One approach might be to install a new mainframe computer large enough to perform the computations. However, mainframes are expensive, and if they're sized to fit today's demand, then it's hard to scale them up to provide more capacity as the number of users grows. Moreover, a mainframe represents a single point of failure in what is otherwise a decentralized network of traders and analysts.

Increasingly, IT departments are turning to a different approach: a client-server system based on the use of a virtual compute server implemented on a number of cooperating workstations or PCs. To use the system, clients submit pricing requests (jobs) to the virtual compute server, where one or more of its processors work to complete the job as quickly as possible. Figure 1 illustrates how such a pricing system might look.

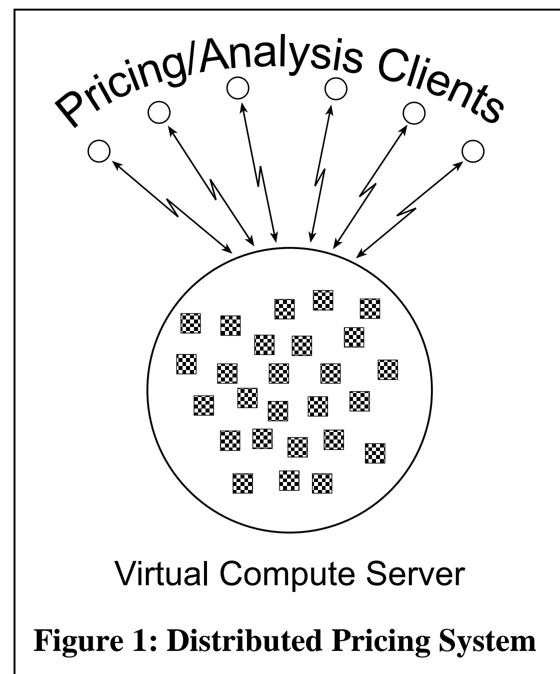

Figure 1: Distributed Pricing System

From an IT perspective, this sort of system is particularly attractive. It is comprised of completely standard commercial hardware and software components, familiar and comfortable to developers, users, and systems administrators alike. Since the computational power of the server derives from an aggregation of machines, it's possible to enhance the system by adding new machines without affecting ongoing activities. The server can even be expanded "on the fly" to meet peaks in demand by adding machines temporarily. Finally, the system can be up and running full time, with hardware maintenance and upgrades handled incrementally, a few machines at a time.

The system is equally beneficial from the clients' point of view. They get:

- *Responsiveness*: Client jobs are completed quickly by dynamically applying multiple processors in parallel to speed up job processing.
- *Reliability*: Client jobs are completed successfully on a 7x24 basis, even if some portions of the virtual compute server or the communications network fail.
- *Flexibility*: More important clients or jobs get faster service than less important ones because the assignment of processors to jobs is demand and priority driven.

You, of course, have the task of building, maintaining, and managing the system, which is a good example of what is known as a *distributed computing application*—one designed to perform a number of things simultaneously at locations that may be physically dispersed. The

pricing system is particularly interesting because it encompasses two related, but distinct, aspects of distributed computing:

1. The traditional interaction between lightweight clients and a centralized server; and
2. The use of parallel processing to perform work on the server itself.

As compared with traditional program development, the task of writing distributed applications leads to a new programming challenge: *coordination among the components of the application*. Coordination encompasses a variety of issues that arise in distributed programming over and above those associated with ordinary programming:

- *communication* (including sharing or exchange of data) among the processes;
- *creation* of multiple execution units (which we'll call *processes*[1]);
- *scheduling* of the processes; and
- *synchronization* among the processes.

Systems and tools for distributed programming must give users convenient ways to deal with all four of these issues. Ideally, developers should be able to design and program at a high level so that their distributed programs can be realized efficiently in a variety of different computational environments, without major redesign or recoding for each one. For example, the overall architecture and most of the code for the distributed pricing system should not depend on whether the clients are PCs or UNIX workstations or whether the virtual compute server is a shared-memory multiprocessor, a distributed-memory parallel computer, or even a networked cluster of workstations or PCs on a local area network.

Not many of the distributed computing tools out there today can do everything you need without introducing a high degree of complexity. One that can is Paradise®[2], Scientific Computing Associates, Inc.'s middleware system based on the concept of *virtual shared memory*. In the remainder of this white paper, we'll discuss virtual shared memory technology and demonstrate how the Paradise system can meet all of the demands of your virtual compute server system.

Virtual Shared Memory

Over the years, a number of different approaches have been offered to deal with distributed programming and the coordination problem. Some of these have been architecture-specific (for example, the use of raw locks or synchronization operations available only on shared-memory machines). Others have been more generic (such as those based on standard libraries or new languages). Almost all, however, have failed to recognize that there are only a few essential distinctions between distributed and traditional programming—exactly the ones that are reflected in the coordination problem. This has usually led to *ad hoc* treatments of the four issues listed above, with the result that distributed programming tools have often been burdened with tremendous complexity.

---

[1] Except as noted in context, we will use the term *process* as a shorthand denoting either a full process or a lightweight thread. The reader should understand, however, that a thread shares resources (e.g. memory, I/O state) with other lightweight threads, whereas a full-fledged process possesses its own separate address space. This distinction may have significant practical implications for application development, but the two models are quite similar with respect to the coordination issues addressed in this white paper.

[2] Linda, Paradise, and Piranha are registered trademarks, and TupleSpace is a trademark, of Scientific Computing Associates, Inc. Other trademarks and registered trademarks used in this document are the properties of their respective owners.

Long experience has shown us that the introduction of a single, easy-to-understand concept—*virtual shared memory* or VSM—is sufficient to endow programmers with all of the power that they need to address the coordination problem and distributed programming. A VSM is a shared object repository that can be used to store data that must be shared among the pieces of a distributed program. It is "virtual" in the sense that no physically-shared memory is required to create a VSM, so VSMs are useful in a variety of computing environments ranging from shared-memory multiprocessor machines to parallel mainframes to workstation or PC clusters. By combining VSMs with suitable application programming interfaces (APIs), it is possible to build distributed programming tools or middleware environments that have the following desirable properties:

- *High Performance*: VSM implementations can be tailored to the particular processor/network architecture used in practice, so they can deliver maximum performance.
- *Ease of Use*: APIs for VSM systems can be very simple, since they need only include a small number of operations (things like *store*, *copy*, and *retrieve*). Such APIs may be implemented as extensions to standard programming-languages, thereby facilitating compile-time preprocessing to catch a wide range of errors and to provide optimized performance of individual operations.
- *Data Persistence and Accessibility*: VSM data is available concurrently to any process that knows about the VSM; moreover, VSMs can function as random access memories, just as ordinary memories do.
- *Fault Tolerance*: VSMs can support transactional semantics, which makes it possible to develop fault-tolerant applications in a particularly straightforward way.
- *Dynamic Scheduling*: VSM-based distributed applications can adapt easily to changes in the processor or job pool that demand dynamic reassignment of resources to meet evolving system requirements.
- *Heterogeneity*: VSMs offer a natural mechanism for transferring data among heterogeneous systems, since the VSM can serve as a platform-independent repository.
- *Load Balancing*: VSMs can use data "pull" semantics, rather than the data "push" semantics more common in other middleware technologies. This makes it extremely easy to load balance among processors, since each processor can take just as much or as little data as it can handle.
- *Manageability*: Since VSMs store objects rather than raw bits, VSM systems can offer visualization and management tools that make it easy to see what is stored in the system and what operations are in progress. A VSM system that makes use of a VSM server to host the individual VSMs can also offer centralized management tools to control the entire system.
- *Security*: From a security perspective, VSMs have many characteristics similar to ordinary file systems. Access to an individual VSM requires a "handle," analogous to a file handle, that is obtained from a system-wide authority (a "handle server"). By limiting access to handles and restricting the capabilities associated with each handle, it is easy to control both who has access to each VSM, and what operations each user may perform.
- *Simple Hardware Enhancement and Upgrade Paths*: For VSM systems that use "pull" semantics, hardware enhancements and upgrades can be extremely simple. Newly added

processors will start to interact as soon as they're connected, increasing the power of the system immediately. Replacement can be accomplished by shutting down a particular processor and installing a new one in its place. Most often, distributed applications need not even know about such changes.

Scientific Computing Associates, Inc. has developed two VSM-based middleware tools, Linda® and Paradise, that support a type of VSM called a TupleSpace™ for distributed computing. Linda and Paradise differ in the way that they implement their VSMs, with the result that Linda is tuned for parallel computing, whereas Paradise is aimed at a broader range of distributed computing applications. However, from a developer's point of view, both provide similar, easy-to-use solutions to the coordination problem based on TupleSpace VSMs. In this paper we'll focus on Paradise and show how it can meet all of the demands of the client-server pricing system.

TupleSpaces and Paradise

TupleSpaces, the special virtual shared memories in SCIENTIFIC's Paradise system, are globally shared, associatively addressed memories in which programs can deposit data objects known as tuples. A tuple is simply a vector of typed values, called fields. Each field may have one of three basic forms: a constant, or an expression that evaluates to a constant, or a formal parameter. Fields may contain either scalars or aggregate types such as arrays or structures (the latter are treated as opaque blocks of binary data). Formal parameter fields, which begin with a question mark (as in "`?v`"), are most often seen in templates used to retrieve data from a TupleSpace—the variable specified is the one that receives the data held in the corresponding field of some tuple in the TupleSpace.

Here are some simple examples using C, for which we assume the following declarations:

```
float f;
char a[20];
struct STYPE s;
```

Tuple	*Fields*
`("int constant", 1)`	literal character string; `int` constant
`("float variable", f)`	literal character string; `float` value
`("array", a)`	literal character string; `char` array of length 20
`("array2",a:10)`	literal character string; `char` array of length 10
`("structure", s)`	literal character string; `struct` of type `STYPE`

The Paradise system is itself a client-server system. The Paradise Server can maintain an unlimited number of TupleSpaces, each of which may be transient (created and destroyed within the lifetime of a single application) or persistent (remaining available until deleted, even if no applications are using it at any given time). Persistent TupleSpaces are especially useful for secure communications among different applications that may not run concurrently. To use a TupleSpace, an application must obtain a *handle* for it from the Paradise Handle Server, and it is possible to control the way that an application uses a TupleSpace by associating a set of capabilities with the handle it receives.

Paradise offers three basic operations (`out`, `in`, and `rd`) to access TupleSpaces, instead of the two (`read` and `write`) that are available for conventional address-based memories. Paradise operations should be viewed as extensions to an underlying programming language (such as C, C++, FORTRAN, or Java, for example). For each supported language, the Paradise system includes a pre-compiler that processes all program statements involving TupleSpaces and tuples and generates a customized set of support routines to perform the Paradise operations at runtime.

The basic forms of the main Paradise operations on tuples are listed in the following table, where `ts` is a handle for a Paradise TupleSpace:

Operation	*Action*
`out@ts()`	Places a tuple into the TupleSpace designated by `ts`
`in@ts()`	Reads values from and removes a tuple in the TupleSpace designated by `ts`
`rd@ts()`	Reads values from a tuple in the TupleSpace designated by `ts`, leaving the tuple in place

The `out` operation takes the fields of a tuple as its arguments. Each field is evaluated, and the resulting tuple is installed in the appropriate TupleSpace.

An `in` or `rd` operation takes the fields of a *template* as its arguments. A template is used both to select a tuple to access and to specify where to copy the data. Like a tuple, a template consists of a sequence of typed fields, some of which may hold values (such fields are known as *actuals*)—either constants or expressions which evaluate to constants—and some of which are formal parameters (known as *formals*) indicating the variables into which the data in the corresponding field of the accessed tuple will be copied. The formal fields begin with a question mark.

Here is a sample template:

```
("simple", ?i)
```

In this template, the first field is an actual, and the second field is a formal. If this template were used as the argument to an `in` or `rd` operation that succeeded in finding a matching tuple, then at the end of the operation, the variable `i` would be assigned the value in the second field of the selected tuple. Formal fields can be used to assign values to variables of any data type of the underlying programming language, including aggregate types.

In an `in` or `rd` operation, Paradise selects the tuple to access by matching the argument template against the tuples in the referenced TupleSpace (the one designated by the handle in the operation). A template matches a tuple when all three of the following conditions hold:

- They both have the same number of fields.
- The types and values of all actuals in the template are the same as those of the corresponding fields in the tuple.
- The types of all formals in the template match the types of the corresponding fields in the tuple.

If more than one tuple matches the template, then the Paradise Server will select one for access.

Paradise's operations go beyond tuple selection and data movement—they also provide for synchronization. An `out` operation will not return until it is safe to modify the local data used to

create the new tuple. An `in` or `rd` operation will block (that is, wait) until a matching tuple is available in TupleSpace. (As alternatives, Paradise also provides the `inp` and `rdp` operations, which are non-blocking variants of `in` and `rd` that return Boolean values to indicate whether a matching tuple is available.)

In addition to the data access operations just discussed, Paradise offers a number of operations to create and destroy TupleSpaces, to manage access to TupleSpaces and TupleSpace handles, to group multiple operations into atomic units, and to handle errors. A number of these, including Paradise transactions, will be addressed in context in later sections.

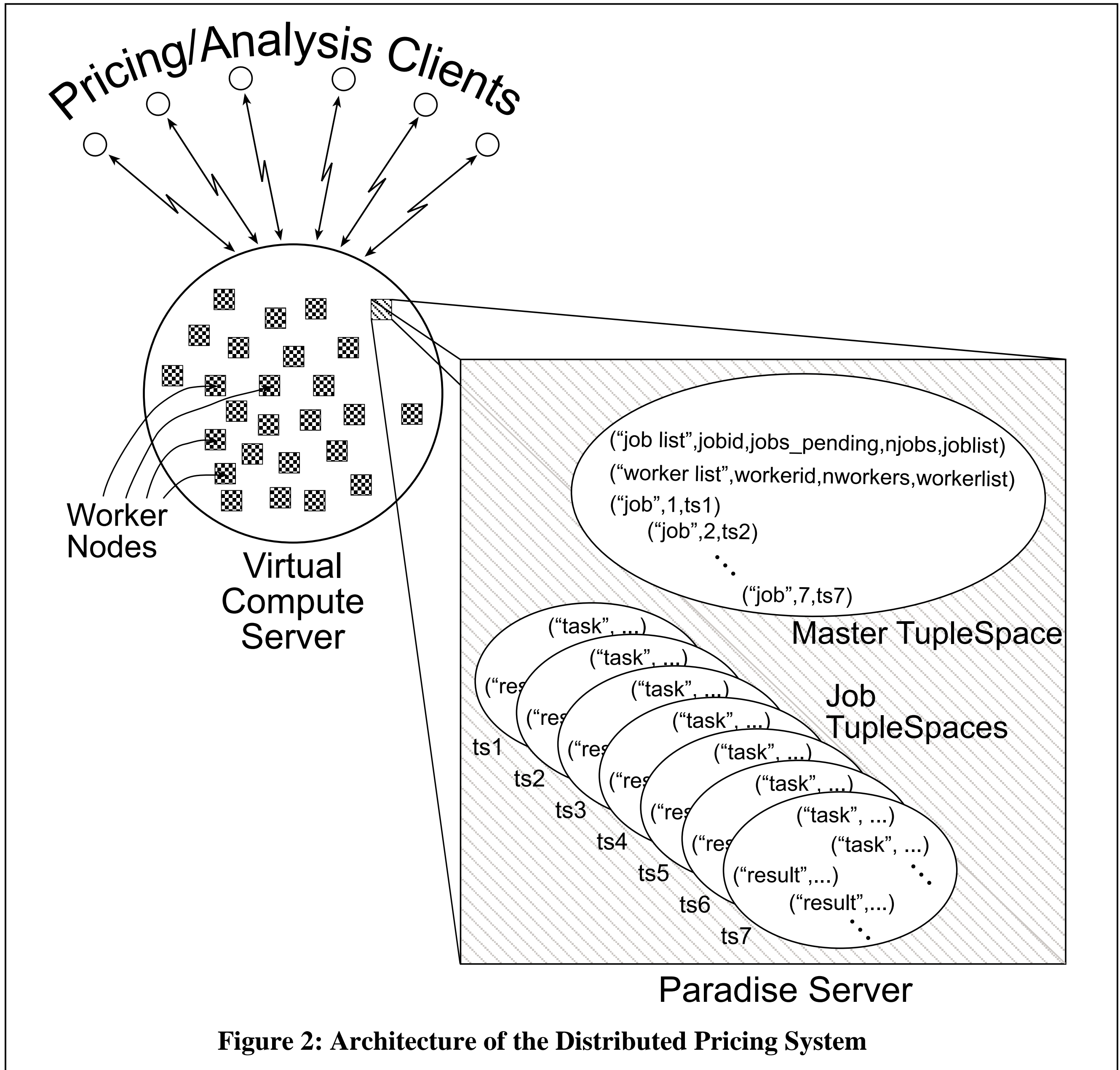

Figure 2: Architecture of the Distributed Pricing System

Implementing the Pricing System

Let's return now to the distributed pricing system that we introduced at the beginning of this white paper to see how you would use Paradise to build that application. The overall architecture of the system, showing the various TupleSpaces that are required, is illustrated in Figure 2.

The basic flow of the system is quite simple. Figures 3 and 4, respectively, contain the top-level client and worker code in an application framework suitable for implementing a client-server system of the sort depicted in Figure 2.

Each client performs a number of steps when it needs service. Beforehand, it divides its job into a number of tasks that can be distributed to individual worker nodes in the virtual server. Then:

1. It creates a new Job TupleSpace for its job data.
2. It enters its job in the list of jobs.
3. It inserts the tasks for its job into its Job TupleSpace.
4. It waits for and collects the results as they appear in its Job TupleSpace.
5. It removes its job from the system.

```
1   int client(int ntasks, TASK tasks[], RESULT results[])
2   {
3     TSHANDLE jobts, masterts;
4     int jobid, i, tasknum;
5     TASK task;
6     RESULT result;

7     paradise_catch(Exit_Handler);

8     waitfor_handle("Pricing System", "MasterTS", &masterts);

9     open@masterts();

10    jobid = addjob(masterts, &jobts);

11    if (jobid == NOJOB) return(NOJOB);

12    for (i = 0; i < ntasks; i++) out@jobts("task", i, tasks[i]);

13    for (i = 0; i < ntasks; i++) {
14      in@jobts("result", ?tasknum, ?result);
15      results[tasknum] = result;
16    }

17    removejob(jobid, masterts, jobts);

18    return(jobid);
19  }
```

Figure 3: The Top-level `client()` Function

```
1   int worker()
2   {
3     int myjobid, tasknum;
4     TSHANDLE myjobts;
5     TASK task;
6     RESULT result;

7     paradise_catch(CATCH_IGN);
8     if (lookup_handle("Pricing System", "MasterTS", &masterts) != 0)
9       if (setup_system(&masterts)) return(-1);

10    catch@masterts(Exit_Handler);
11    open@masterts();

12    myworkerid = addworker(masterts);

13    if (myworkerid == -1) return(-2);

14    while(1) {
15      setjmp(Get_Job);

16      myjobid = selectjob(myworkerid, masterts, &myjobts);

17      catch@myjobts(EOJ_Handler);
18      while (1) {
19        xaction@myjobts();
20          in@myjobts("task", ?tasknum, ?task);
21          compute(task, &result);
22          out@myjobts("result", tasknum, result);
23        commit@myjobts();
24      }
25    }
26  }
```

Figure 4: The Top-level `worker()` Function

Concurrently, each worker node in the Virtual Compute Server performs the following steps:

1. It checks the list of active jobs to select a job to work on. In the example described here, it simply selects a job being worked on by the smallest number of workers.
2. It repeatedly selects a task from the selected job's Job TupleSpace, performs the required computation, and returns the result to the Job TupleSpace.
3. When all the tasks for the selected job are complete, it returns to select another job.

It would be reasonably straightforward to generalize the behaviors of the clients or workers in a number of ways (e.g., by permitting worker nodes to switch among pending jobs under various conditions). However, this isn't required in order to illustrate the virtues of VSM systems like Paradise.

As illustrated in Figure 2, there are two system management tuples that reside in the Master TupleSpace, the TupleSpace used to manage the entire pricing system system. The entire operation of the distributed pricing system is controlled using these tuples:

- The *Job List Tuple*, a tuple containing the following four items describing jobs:
  1. `jobid`: an `int` used to assign a unique identifier to each job;
  2. `jobs_pending`: an `int` used as a Boolean to indicate whether there are any active jobs currently in the system;
  3. `njobs`: an `int` whose value is the number of active jobs currently in the system; and
  4. `joblist`: an array of `structs` of type `JOB` containing one entry per active job.
- The *Worker List Tuple*, a tuple containing the following three items describing workers:
  1. `workerid`: an `int` used to assign a unique identifier to each worker;
  2. `nworkers`: an `int` whose value is the number of active workers currently in the system; and
  3. `workerlist`: an array of `structs` of type `WORKER` containing one entry per active worker.

During normal operation, the data structures in these tuples are manipulated using a number of service functions, including:

- `addjob()`: a function that adds a new job to the `joblist`;
- `removejob()`and `cleanupjob()`: functions that remove a job from the `joblist`;
- `addworker()`: a function that adds a new worker to the `workerlist`;
- `removeworker()`: a function that removes a worker from the `workerlist`;
- `setstatus()`: a function that updates a worker's status in the `workerlist`; and
- `selectjob()`: a function that examines the list of active jobs and the status of all the workers and selects a job for a worker to work on.

In addition, an initialization function `setup_system()` is used to create the control tuples when the first worker node begins operation.

It is worth pointing out that most of the code discussed here is generic—it could serve as the basis for any client-server system, not just the distributed pricing system. The pricing-specific aspects of the code are encapsulated in two data arrays and the actual computation function:

- `tasks`: an array of dimension `ntasks` containing `structs` of type `TASK` specifying the data for the tasks in a job;
- `results`: an array of dimension `ntasks` containing `structs` of type `RESULT` that receive the results for the tasks;
- `compute()`: a function that takes as input a `struct` of type `TASK` and returns as output the computed result as a `struct` of type `RESULT`.

Only these components would require redefinition to use the framework for other applications.

Now let's examine the application framework in detail, beginning with the client.

Client Processing

The `client()` function is designed to submit and manage new pricing jobs on behalf of a user of the pricing system. It takes three arguments:

1. the number of tasks in the job (`ntasks`),
2. a pointer to the array of tasks (`tasks`), and
3. a pointer to the array of results (`results`).

The `client()` function begins by specifying the default mechanism for handling any errors that might occur. For example, if the entire distributed pricing system were to be shut down, the client would want to catch the error and take appropriate action (such as printing a message). The `paradise_catch` statement in line 7 establishes the function `Exit_Handler()`(not shown) as the client's default error handler for the Paradise system. Later on, we'll see that Paradise enables different error handlers to be specified for each TupleSpace handle.

The first step in job submission is to locate the Master TupleSpace for the distributed pricing system. This is accomplished using the Paradise `waitfor_handle` function (line 8), which queries the Paradise Handle Server to obtain the TupleSpace handle for the Master TupleSpace. (When the pricing system starts up, the Master TupleSpace is created and registered with the Handle Server. If the pricing system is not running when the client starts operation, the `waitfor_handle` function will block and wait until the system is started.) The variable `masterts` is used to store the handle, which must be opened (line 9) before use.

Next, the client invokes the function `addjob()` (shown in Figure 5), passing it the open `masterts` handle. `addjob()` is responsible for modifying the job list tuple and taking other steps to submit the new job to the system. Upon successful completion, it returns the unique job identifier assigned to the new job and an open handle for the Job TupleSpace to be used for the `TASKS` and `RESULTS` associated with the new job. If `addjob()` is unable to submit the new job, then it returns a special job identifier (`NOJOB`) so that the client can respond appropriately.

`addjob()` uses an `in` operation (line 6) to obtain the job list tuple, which it requires in order to perform its work. Notice the use of a Paradise transaction (beginning with the `xaction` statement in line 5) to ensure the safety of this critical tuple. Paradise transactions provide a general way of treating a sequence of Paradise tuple operations as if it were a single atomic operation. Either all of the operations will succeed, or the state of the referenced TupleSpace will be rolled back to appear as if none of the operations had taken place. In this instance, by using a transaction, the client can guarantee that the critical system-wide data in the job list tuple will not be lost even if the client were to fail while manipulating it.

Once `addjob()` has the job list tuple, it must deal with the possibility that the new job will exceed the capacity of the distributed pricing system (lines 7-10). If the system is unable to accommodate an additional job, then `addjob()`cancels the open transaction (using the `cancel` operation in line 8) and returns immediately. The effect of the `cancel` operation is to return the job list tuple to the Master TupleSpace, as if it had never been removed.

The following four lines of `addjob()` (lines 11-14) assign a unique job identifier to the new job and fill in an unused entry of the `joblist` array with information about the job. In our simple example, the entries of the array contain only the `jobid` and the number of workers currently working on the job (initially 0).

```
1   int addjob(TSHANDLE masterts, TSHANDLE *jobts)
2   {
3     int myjobid, jobid, jobs_pending, njobs;
4     extern JOB joblist[MAXJOBS];

5     xaction@masterts();
6       in@masterts("job list", ?jobid, ?jobs_pending, ?njobs, ?joblist:);

7       if (njobs == MAXJOBS) {
8         cancel@masterts();
9         return(NOJOB);
10      }

11      myjobid = ++jobid;
12      joblist[njobs].id = myjobid;
13      joblist[njobs].workers = 0;
14      njobs++;

15      (*jobts) = create@rootts(PARADISE_DESTROYONCLOSE);
16      out@masterts("job", myjobid, (*jobts));

17      out@masterts("job list", jobid, JOBPENDING, njobs, joblist:);
18    commit@masterts();

19    return(myjobid);
20  }
```

Figure 5: The addjob() Function

Next, `addjob()`creates a new Job TupleSpace to hold the tasks and results for the new job. The Paradise Server maintains a hierarchy of TupleSpaces, rooted in the special system TupleSpace called `rootts`; the new Job TupleSpace is created as a child of `rootts`. The Paradise `create` operation returns an open TupleSpace handle, so the `client()` function will be able to use it immediately after `addjob()` returns. The modifier `PARADISE_DESTROYONCLOSE` is used to tell Paradise that the new TupleSpace should be destroyed whenever the handle is closed.[3]

Once it has obtained a `jobid` and created a Job TupleSpace, `addjob()` can update information in the Master TupleSpace to indicate the presence of the new job. First, in line 16, it creates a job tuple containing the `jobid` and a handle for the newly created Job TupleSpace. (TupleSpace handles can be included in tuples just like any other data; Paradise inserts a closed copy of the handle when it creates the tuple.) Then, using the `out` operation in line 17, it places the updated job list tuple in the Master TupleSpace. Finally, it commits its open Paradise transaction, thereby making the new job tuple and the updated job list tuple available for use by other processes.

Assuming that the `addjob()` function successfully submits the new job, the `client()` function continues by first populating the new Job TupleSpace with all of the task descriptor tuples (line 12) and then retrieving the job's results in the loop in lines 13-16. Transactions aren't used for these loops, since the tuples are specific to this particular job and don't impact overall system

[3] `PARADISE_DESTROYONCLOSE` is a modifier first implemented in version 6.0 of Paradise. We'll see that it is useful here to ensure proper operation of the pricing system if the client fails or is terminated abruptly by the user.

operation. One important detail to note is that the result collection loop does not assume any particular ordering of the returned results. This allows for the possibility that some tasks may be more time consuming than others or that not all the worker nodes operate at the same speed.

When all of the results have been collected, the client cleans up by invoking the `removejob()` function (line 17) and returning. The `removejob()` function (shown in Figure 6) begins by updating the control information in the Master TupleSpace. It retrieves the job list tuple (line 7), deletes its job from the job list (implemented in the `cleanupjob()` function that isn't shown), removes the client's job tuple (line 9), and returns the job list tuple containing the updated `joblist` array and a properly set `jobs_pending` field (lines 10-11). Once these steps are complete, `removejob()` destroys the client's Job TupleSpace (line 12). As in the `addjob()` function, a Paradise transaction is used to protect the critical system-wide data.

There are two points to be made about the operations performed by `removejob()`. First, both `in` operations (lines 7 and 9) make use of a type of field known as an "anonymous formal." Anonymous formals (which have the form "`?type`") are similar to other formals except that they cause Paradise to discard the corresponding tuple field instead of copying it to a local variable. They are useful for matching the type of a field without either matching or copying its value.

The other point concerns the `close` operation in line 12. This operation uses the modifier `PARADISE_DESTROY` to tell Paradise to destroy the TupleSpace immediately and force any pending operations to fail. This is important for correct operation of the distributed pricing system because, as we will see below, the error is used to cause the worker nodes that had been working on this job to select new jobs to work on. Because the Job TupleSpace was created in `addjob()`using the `PARADISE_DESTROYONCLOSE` modifier, it will be destroyed automatically by the Paradise Server should the client fail before executing this `close` operation.

```
1   void removejob(int myjobid, TSHANDLE masterts, TSHANDLE jobts)
2   {
3     int jobid, njobs;
4     int i, j;
5     extern JOB joblist[MAXJOBS];

6     xaction@masterts();
7       in@masterts("job list", ?jobid, ?int, ?njobs, ?joblist:);

8       cleanupjob(myjobid, &njobs, joblist);
9       in@masterts("job", myjobid, ?TSHANDLE);

10      if (njobs == 0)
          out@masterts("job list", jobid, NOJOB, njobs, joblist:);
11      else out@masterts("job list", jobid, JOBPENDING, njobs, joblist:);

12      close@jobts(PARADISE_DESTROY);
13    commit@masterts();

14    return;
15  }
```

Figure 6: The removejob() Function

Worker Processing

We turn now to the workers in the distributed pricing system. The overall architecture of a worker includes a `main()` routine that calls the `worker()` function shown in Figure 4. The `main()` routine itself may be started either manually by an administrator (using the ordinary command line interface for a particular machine, for example) or automatically by an initialization program for the entire pricing system application. The latter case is implemented using the `paradise_spawn` operation that Paradise provides to allow one program to start another program on a specified processor.

The workers in the distributed pricing system are expected to run continuously over an extended period of time. As a result, it is very important for them to handle errors in a robust way, since failure to do so would impact the entire system, not just a single client. The client implementation used a default error handler for the entire client application, with the idea that all errors would be treated in the same way—most likely by printing a suitable message and exiting.

The workers, however, must use a more sophisticated approach. Paradise facilitates this by providing a general error handling capability that allows different error handlers to be specified for each TupleSpace handle. In the absence of a specific handler, the default action is for Paradise to print an error message and cause the program to exit. As discussed above, the `paradise_catch` operation can be used to change this default behavior, and the operation in line 7 of the `worker()` function instructs Paradise to ignore errors so that they can be handled by the program itself.

As a first cut from a worker's point of view, one might divide errors into three main types: local failures on the worker's node, Paradise Server failures, and failures related to specific TupleSpaces on the server. Local errors (such as processor or disk failures) are likely to be catastrophic to the worker, so the worker won't try to recover from them. However, it will use Paradise transactions to ensure that local failures won't affect the operation of the distributed pricing system as a whole.

Paradise Server failures (either hardware or software) might be handled in a variety of ways—for example, by locating an alternative server. However, for simplicity in the presentation here, we will assume that a worker should exit if the Paradise Server fails.

Finally, that leaves errors arising in operations on the Master or Job TupleSpaces in the distributed pricing system. A worker will actually treat these as normal events and use them to control its overall operation:

- If an error occurs in an operation on the Master TupleSpace, the worker will assume that either the server has failed or that the distributed pricing system has shut down and destroyed the Master TupleSpace. In either event, the worker will exit. The handler for this case is the `Exit_Handler()` function, which is simple enough that it is not shown.

- On the other hand, if an error occurs in an operation on a Job TupleSpace, the worker will assume that the server is still operational, but that the Job TupleSpace has been destroyed by the client which submitted the job. This simply indicates normal completion of the job, and the worker will simply select a new job to work on. (If the error is actually due to failure of the Paradise Server, this will become evident when the worker tries to select a new job.) The handler for this case is the `EOJ_Handler()` function, which is shown in Figure 8 and discussed in detail below.

To begin its normal processing, the worker tries to look up a handle for the Master TupleSpace using Paradise's `lookup_handle()` function (line 8). If this succeeds, `lookup_handle()` will return a zero value, and `masterts` will contain a closed handle for the Master TupleSpace. If the lookup operation fails, `lookup_handle()` will return a nonzero value, which will cause the worker to invoke the function `setup_system()` to initialize the distributed pricing system. If the initialization process fails, then the worker will return an error code of –1 to its `main()` routine, which will then exit.

Assuming that the worker can successfully obtain a handle for the Master TupleSpace, the next step is to set up the error handler for that handle. The worker does this using the `catch` operation in line 10 of the `worker()` function. This tells Paradise to invoke the `Exit_Handler()` function if an error occurs on any operation involving the `masterts` handle. We have omitted the code for this routine, but it might simply print a message and exit.

Following the `catch` operation, the worker opens the `masterts` handle (line 11) and invokes the `addworker()` function (shown in Figure 7) to register itself as a new worker in the distributed pricing system. If the registration fails (indicated by a return value of –1), then `worker()` returns with an error code of –2 (line 13).

```
1   int addworker(TSHANDLE masterts)
2   {
3     int myworkerid, workerid, nworkers;
4     extern WORKER workerlist[MAXWORKERS];

5     xaction@masterts();
6       in@masterts("worker list", ?workerid, ?nworkers, ?workerlist:);
7       if (nworkers >= MAXWORKERS) {
8         cancel@masterts();
9         return(-1);
10      }

11      myworkerid = ++workerid;

12      workerlist[nworkers].id = myworkerid;
13      workerlist[nworkers].status = IDLE;
14      workerlist[nworkers].job = NOJOB;
15      nworkers++;

16      out@masterts("worker list", workerid, nworkers, workerlist:);
17    commit@masterts();

18    return(myworkerid);
19  }
```

Figure 7: The addworker() Function

The `addworker()` function (Figure 7) is structured in much the same way as the `addjob()` function (Figure 5). In lines 5-6, it starts a Paradise transaction and retrieves the worker list tuple using an `in` operation. It then checks to see if the distributed pricing system can handle an additional active worker (lines 7-10), and if it can't, it cancels the transaction and returns an error

code of -1. Otherwise, it claims the next worker identifier (line 11) and initializes an unused entry of the worker list for the new worker (lines 12-15). Finally, it replaces the worker list tuple in the Master TupleSpace, commits the transaction, and returns the new worker's `workerid`.

If the registration in `addworker()` succeeds, the worker continues by entering the main loop of the `worker()` function (lines 14-25 of Figure 4). The strategy here is that the worker will remain in this loop forever until one of its operations on the Master TupleSpace fails, causing invocation of the `Exit_Handler()` error handler and a graceful exit. This approach is reasonable because there are only two causes for such operations to fail: failure of the Paradise Server or removal of the Master TupleSpace. In either case, there is no reason for the worker to continue operation since the entire distributed pricing system must have shut down.

The worker's main `while` loop repeatedly selects a job to work on (line 16) and performs computation for the selected job (lines 18-24) until that job is complete. The computation loop is a perpetual loop in which the worker repeatedly retrieves an arbitrary task tuple from the Job TupleSpace, uses the `compute()` function to perform the necessary computations, and returns a result tuple to the Job TupleSpace. These operations are wrapped in a Paradise transaction in order to guarantee that the task tuple will be reinstated should the worker fail to complete it.

As we noted earlier, the completion of the job will be indicated to the worker by failure of a Paradise operation (in this case the `in` operation in line 20) following the client's destruction of the Job TupleSpace for the job. When that happens, the worker is supposed to continue by selecting a new job. This error handling behavior is achieved using the `setjmp()` and `longjmp()` operations of C in combination with Paradise's error handling facility. The worker first performs a `setjmp()` at line 15 to set the location at which processing should continue after error handling is complete. Then, after obtaining a handle for the proper Job TupleSpace from the invocation of `selectjob()` in line 16, it uses a `catch` operation to associate the `EOJ_Handler()` function as the error handler for that handle. By examining Figure 8, we can see that when an error occurs, the `EOJ_Handler()` function will close the `jobts` TupleSpace handle, modify the worker list to mark the worker as idle, and perform a `longjmp()` to return to the top of the worker's main `while` loop.

```
1   void EOJ_Handler(TSHANDLE *jobts,
                     int ecode, int opcode, int line,char *file)
2   {
3     int workerid, nworkers;
4     WORKER workerlist[MAXWORKERS];

5     catch@(*jobts)(CATCH_IGN);
6     close@(*jobts)();

7     xaction@masterts();
8       in@masterts("worker list", ?workerid, ?nworkers, ?workerlist:);
9       setstatus(myworkerid, IDLE, NOJOB, nworkers, workerlist);
10      out@masterts("worker list", workerid, nworkers, workerlist:);
11    commit@masterts();

12    longjmp(Get_Job, 1);
13  }
```

Figure 8: The `EOJ_Handler()` Error Handler

To complete our discussion of the worker, we turn to the `selectjob()` function shown in Figure 9. The purpose of `selectjob()` is to find a suitable job for the worker. In principle, there are many criteria that might go into such a selection, including the nature of the job, the capabilities of the worker, and the activities of other workers, to name a few. In this example, however, we've used a very simple criterion: select a job on which the smallest number of workers is already working.

In order to make this selection, `selectjob()`must examine and modify both the job list tuple and the worker list tuple. It must also account for the possibility of jobs that are "bad" in the sense that the clients that created them have died without cleaning up some of the control tuples in the Master TupleSpace. (This could happen if a client machine failed, for example. In that event, the client's Job TupleSpace would be destroyed automatically since it was created using the `PARADISE_DESTROYONCLOSE` modifier; but the corresponding job tuple would not be removed, nor would the job information in the job list tuple be updated.)

A single transaction is used to protect all of the Paradise operations in `selectjob()`. Lines 9 and 10 perform `in` operations to retrieve the job list and worker list tuples. The actual value "`JOBPENDING`" is used in the `jobs_pending` field of the template in the first `in` operation in order to force `selectjob()` to block there until there is at least one active job in the system. This avoids the situation in which the worker repeatedly retrieves the job list only to find out that there are no jobs to work on.

Lines 12-20 use the selection criterion discussed above to select a job from among the `njobs` currently active jobs. Once a job has been chosen, `selectjob()` performs a `rd` operation to obtain a handle for that job's Job TupleSpace (line 21) and checks the validity of the job by attempting to open the handle (line 23).

If the `open` operation fails, then the job is "bad," and `selectjob()` performs appropriate cleanup (lines 24-26). Otherwise, it updates the job list and worker list to indicate its selection (lines 29-30). In either case, `selectjob()` reaches the bottom of the `do` loop at line 32, where it checks to see if it has selected a valid job. If so, then it exits from the loop, replaces the job list and worker list tuples in the Master TupleSpace, commits its transaction, and exits.

If the selected job was "bad," then `selectjob()` attempts to make a new selection provided that there is at least one active job still remaining. If the "bad" job was the last active job, then `selectjob()`replaces the job list and worker list tuples in the Master TupleSpace, commits its transaction, and returns to the `in` operation at line 8 to await the appearance of a new active job. (The control tuples must be replaced in order to permit clients to submit new jobs.)

```
1   int selectjob(int myworkerid, TSHANDLE masterts, TSHANDLE *myjobts)
2   {
3     int jobid, njobs, workerid, nworkers;
4     extern JOB joblist[MAXJOBS];
5     extern WORKER workerlist[MAXWORKERS];

6     int myjobid, jobindex, mincount, i, jobs_pending;

7     do {
8       xaction@masterts();
9         in@masterts("job list", ?jobid, JOBPENDING, ?njobs, ?joblist:);
10        in@masterts("worker list", ?workerid, ?nworkers, ?workerlist:);

11        do {
12          jobindex = 0;
13          mincount = nworkers + 1;
14          for (i = 0; i < njobs; i++) {
15            if (joblist[i].workers < mincount) {
16              mincount = joblist[i].workers;
17              jobindex = i;
18            }
19          }

20          myjobid = joblist[jobindex].id;

21          rd@masterts("job", myjobid, ?(*myjobts));

22          catch@(*myjobts)(CATCH_IGN);
23          if (open@(*myjobts)() != 0) {
24            cleanupjob(myjobid, &njobs, joblist);
25            in@masterts("job", myjobid, ?TSHANDLE);
26            myjobid = NOJOB;
27          }
28          else {
29            joblist[jobindex].workers++;
30            setstatus(myworkerid, ACTIVE, myjobid, nworkers, workerlist);
31          }
32        } while ((myjobid == NOJOB) && (njobs > 0));

33        if (njobs > 0)
34        out@masterts("job list", jobid, JOBPENDING, njobs, joblist:);
35        else out@masterts("job list", jobid, NOJOB, njobs, joblist:);

36        out@masterts("worker list", workerid, nworkers, workerlist:);
37      commit@masterts();
38    } while (myjobid == NOJOB);

39    return(myjobid);
40  }
```

Figure 9: The selectjob() Function

Concluding Remarks

It is clear that building distributed applications is an intellectually challenging activity. Even the relatively simple system we've discussed here has numerous complications:

- *Variable Task Computation Times*: The system has to deal with the fact that variations in task difficulty or worker speed will cause some tasks to take longer others. This design handles this by avoiding ordering constraints on result collection by the clients and by using a decentralized task-pulling strategy rather than preassignment to allocate tasks to the workers.
- *Intrinsic Scalability*: The system is designed to be completely scalable with respect to both clients and servers. Because all of the tuple data accessible to any client or worker who can connect to the Paradise Server, it is possible to add either clients or workers "on the fly" without changing or reconfiguring the system in any way. This is a major advantage of VSM systems over systems based on point-to-point data sharing protocols.
- *Manageability*: While we haven't discussed this in any detail, it is certainly the case that one could easily write a monitoring and management application that would report on the status of the distributed pricing system based on the data available in the Master TupleSpace. The states of jobs and workers are reflected accurately in the job list and worker list tuples, respectively.
- *Reliability and Fault Tolerance*: The use of Paradise transactions enables the distributed pricing system to be tolerant of the most common failures: local failures of either clients (the system continues and workers select new jobs) or workers (only the failed worker is affected). With only a little more work it would have been possible to deal with failures of the Paradise Server as well (by switching to an alternative server).

Clearly good programming tools and systems can ease the process of developing distributed applications. A large number of real-world users have found that virtual shared memory systems in general, and SCIENTIFIC's commercial Linda and Paradise systems in particular, offer the right mixture of high performance, robustness, and ease of use to enable them to build and deploy mission-critical, enterprise-wide distributed applications quickly and correctly. To find out how you can apply SCIENTIFIC's software products to make parallel and distributed computing work for you, contact SCIENTIFIC's sales department:

Sales Department
Scientific Computing Associates, Inc.
One Century Tower
265 Church Street
New Haven, CT 06510

Telephone: (203) 777-7442
Facsimile: (203) 776-4074
E-mail: sales@sca.com
Internet: www.sca.com