SCIDDLE

A tool for large scale distributed computing

Report**Author(s):**

Arbenz, Peter

Publication date:

1994

Permanent link:

<https://doi.org/10.3929/ethz-a-000944414>

Rights / license:

In Copyright - Non-Commercial Use Permitted

Originally published in:

ETH, Eidgenössische Technische Hochschule Zürich, Departement Informatik, Institut für Wissenschaftliches Rechnen 213

Peter Arbenz
Christoph Sprenger
Hans Peter Lüthi
Stefan Vogel

**SCIDDLE: A
Tool for Large
Scale Distributed
Computing**

March 1994

Peter Arbenz
Institut für Wissenschaftliches Rechnen
Eidgenössische Technische Hochschule
CH-8092 Zürich
e-mail: arbenz@inf.ethz.ch

Christoph Sprenger
Institut für Wissenschaftliches Rechnen
Eidgenössische Technische Hochschule
CH-8092 Zürich
e-mail: sprenger@inf.ethz.ch

Hans Peter Lüthi
Interdisziplinäres Projektzentrum für Supercomputing
Eidgenössische Technische Hochschule
CH-8092 Zürich
e-mail: luethi@ips.id.ethz.ch

Stefan Vogel
Interdisziplinäres Projektzentrum für Supercomputing
Eidgenössische Technische Hochschule
CH-8092 Zürich
e-mail: vogel@physik.unizh.ch

This report is also available via anonymous ftp from [ftp.inf.ethz.ch](ftp://ftp.inf.ethz.ch/doc/tech-reports/1994/213.ps)
as `doc/tech-reports/1994/213.ps`.

SCIDDLE: A Tool for Large Scale Distributed Computing

Peter Arbenz and Christoph Sprenger

*Institut für Wissenschaftliches Rechnen
Eidgenössische Technische Hochschule
CH-8092 Zürich*

Hans Peter Lüthi and Stefan Vogel*

*Interdisziplinäres Projektzentrum für Supercomputing
Eidgenössische Technische Hochschule
CH-8092 Zürich*

Abstract

We report on a portable, easily usable communication environment, ‘SCIDDLE’, for distributing computations over heterogenous networks of UNIX computers. SCIDDLE is based on the client-server model. It was designed to support the distribution of large scale numerical computations and to keep its usage as simple as possible. A convenient array handling has been implemented. We demonstrate the usefulness of the system with an application from quantum chemistry on internet-connected workstations and supercomputers.

Keywords. supercomputing, distributed-concurrent computing, heterogeneous network, remote procedure call, client-server model, quantum chemistry

1 Introduction

In recent years it has become evident that, due to constraints of fundamental physical limitations imposed by the speed of light, supercomputers will have to be multiprocessor computers with very many processors in order to achieve substantial performance increases in the near future. Processors of such computers will have local memory and will be tightly coupled, i.e. will be interconnected with networks of very high bandwidth. While such massive parallel processors provide the compute power necessary to attack grand challenge applications which have extremely high needs for compute power, memory space and connection bandwidth, less demanding applications can perform at very respectable speeds on ‘parallel computers’ with powerful but only loosely coupled processors. Such computers are most often formed by network connected workstations and sometimes supercomputers.

According to Amdahl’s law an application to be running successfully with high efficiency on loosely coupled machines necessarily has to be embarrassingly parallel, i.e. the ratio of volumes of communication and computation must be close to zero.

Although tightly and loosely coupled computers behave completely differently, their programming is essentially the same. In both approaches compute performance is gained by distributing applications over several processors. This makes it necessary that different processes belonging to a certain application be able to communicate i.e. exchange data with each other. The processes may or may not reside on the same processor.

*Present address: Rechenzentrum, Universität Zürich-Irchel, Winterthurerstrasse 190, CH-8057 Zürich

As most computer languages do not provide means to create parallel processes let alone connections between them, many attempts have been made to extend existing languages (most notably C and Fortran) by constructs which make it relatively easy for the programmer to write and run distributed computer applications.

Tools to start and monitor distributed applications are usually integrated into these systems. Many of these ‘communication environments for distributed computing’ provide constructs to facilitate message passing (PVM [26], P4 [5],[6], PARMACS [13], Parform [7, 8]). Another environment, Network Linda [9], provides a virtual shared memory called ‘tuple space’ through which data is transferred between processes or from which idle processors can fetch processes (tasks) to be executed.

High Performance Fortran [11] and Vienna Fortran [10] are based on a data-parallel approach. Both provide extensions to Fortran which incorporate vector and matrix types and most importantly syntax constructs which enable the programmer to explicitly distribute the arrays over the processors. As in conventional sequential programs, HP or Vienna Fortran programs have only one logical stream of control.

SCIDDLE is a remote procedure call (RPC) system [3], [23]. It is based on the client-server model. A process, the *client*, can communicate with another process, the *server*, by calling subroutines. A client requests a service and a server delivers it. Message passing is hidden within subroutine calls. The SCIDDLE stub compiler generates the necessary communication routines. The underlying message passing remains transparent to the user.

The client-server model restricts the set of possible communication patterns to the shape of a tree. A client may have multiple servers, but servers cannot communicate with each other. As a consequence, only so-called embarrassingly parallel algorithms can be implemented reasonably with SCIDDLE. In view of Amdahl’s law for parallel computation, we are convinced that only *such* applications are worthwhile (in terms of speedup) to be run on networks of loosely coupled computers.

SCIDDLE was designed to support the distribution of large scale numerical computations and to keep its usage as simple as possible. As all interprocess communication takes place through procedure calls, the user moves within paradigms well-known from sequential programming. The server program is (unless it is client to another server process) identical to the corresponding part of the sequential version of the program. Large-scale means that we assume that the amount of data residing on a processor is close to the physical limits. The buffer space available for building messages is limited and certainly much smaller than the largest message. This means that in general no complete messages are stored.

SCIDDLE is written in C. It permits computations in heterogeneous networks of UNIX computers and can be used in connection with Fortran and C programs. In particular, a convenient array handling has been implemented to transfer submatrices with a regular index range. An additional feature that is not found in other RPC environments is the possibility to save data for reuse in later remote subroutine calls.

SCIDDLE applications run on a heterogeneous network of UNIX machines. Three types of heterogeneity are supported: (a) machine architecture, (b) network and (c) programming language heterogeneity. Currently, SCIDDLE is installed on the following compute platforms: Sun SPARCstation, IBM RS 6000, Silicon Graphics, Cray Y-MP, Convex and NEC SX-3 where it is actively used in quantum chemistry research. These machines may be interconnected by Ethernet, Internet or fiber optic networks. SCIDDLE interfaces with user code written in either C (K&R, ANSI, C++) or Fortran 77.

The outline of the paper is as follows. In section 2 we discuss the client-server model and ways to implement remote procedure calls. In section 3 we give a brief overview on the software components involved in a SCIDDLE application. A more detailed discussion follows in sections 4 to 6. In section 4 we introduce the SCIDDLE remote interface definition language which enables the SCIDDLE compiler to generate server and client stubs. Sections 5 and 6 deal with the most important functions provided by the user library and the run-time system. In section 7 we demonstrate how easily the SCIDDLE system is used by means of an illustrative example. In the final section 8 we report on the implementation of a distributed-concurrent direct SCF scheme on clusters of workstations and supercomputers [16].

2 Programming Model

The programming model used in SCIDDLE is that of remote procedure calls, which is based on the client-server model for process interaction.

2.1 Client-Server Model

In this model, control is distributed among various processes. Processes are classified as either clients or servers. To request a service, a process called *client* sends a request to a *server* process, which then does the work and sends back the response. Each server provides a particular *service*. It is useful to distinguish between services and servers. A *service* is a software entity running on one or more machines. A *server* is the service software running on one machine [12]. In contrast to most RPC systems, where several clients compete for servers, SCIDDLE servers are private to their clients [28]. Multiple requests are queued at the server and processed iteratively. In SCIDDLE the lifetime of a server is constrained by the lifetime of its client. As a consequence of the client-server model, the set of possible communication topologies for a distributed SCIDDLE application is restricted to tree structures.

2.2 Remote Procedure Calls

Remote procedure call (RPC) is a language-level approach to the client-server model based on the fundamental concept of the procedure call. The idea is very simple and arises from the observation that the client-server model for process interaction requires two messages to be exchanged. The client sends off a request and blocks while waiting for a response, transferring control and data to the server. This is very similar to what happens in a local procedure call. Because procedure calls are a well-known and well-understood abstraction for structuring sequential programs, it is an obvious idea to use the same concept for distributed programs. Indeed, the goal is to make the use of RPCs as transparent to the application as local procedure calls.

A RPC facility transforms local procedure calls into request and response messages exchanged between the client and server processes: the remote procedure call is sent by the calling process (client) to the called process (server) in the form of a call (request) message containing a procedure identifier and the arguments to be passed. The server executes the procedure and sends back a result (response) message containing the result values and a status code indicating success or failure. The entities performing this transformation are called the *stubs*. Analogous to local procedure calls the caller is blocked while waiting for results. The stubs are the only components aware that a call is remote. There is a stub for each side. They execute in the address spaces of the client and server respectively. Together with the communication system for sending and receiving messages, they form the components involved in a RPC. Fig. 1 illustrates these components schematically. A more detailed treatment of RPCs can be found in [4, 14, 19].

While appearing simple and clean at the first glance, there are a number of problems associated with making RPCs transparent. They arise from the fact that the client and the server reside in different address spaces and are usually located on different machines. We just want to point out some of these issues which complicate remote calls. For a more detailed overview and examples, see [24]. Among these problems are parameter passing, exception handling and data representation.

Passing value parameters to a remote procedure imposes no problems. More difficult are reference parameter or pointers. A reference or a pointer has no meaning in a different address space. SCIDDLE (and most other RPC systems) solves the problem by disallowing references and pointers and passing parameters by-value-result, which means that they are copied in the respective directions. When executing a remote procedure, the chance of something going wrong is greater compared with a local procedure. SCIDDLE RPC provides *at-most-once* call semantics¹. This means that a call was executed exactly once when it terminated normally. In the presence of a failure a call may or may not have been executed. This is similar to local call semantics. In a heterogeneous environment with different machine architectures data compatibility problems occur. To overcome these problems some conversion between different data representations is needed. We

¹For a classification of RPC call semantics, see [22]

Figure 1: Remote Procedure Call (RPC)

use a standard external representation, XDR [25], and convert from native host format to this representation and vice versa.

Figure 2: Control flow for (a) synchronous and (b) asynchronous RPCs

2.3 Asynchronous RPC

There are several problems with ordinary, synchronous RPC [27]. There are conceptual problems inherent to the underlying client-server model as well as transparency, reliability and performance problems. The latter stem from the lack of parallelism. While a synchronous RPC is in progress, the client is always blocked waiting for the response. Client and server are in fact coroutines. A severe efficiency problem occurs, if several independent and time-consuming invocations could be processed in parallel. A possible solution to overcome this problem is to create separate threads of control for each call on the client side. But first, this leads to an increased complexity of the programming model and second, there is no thread package available under UNIX that is (a) portable and (b) does not block the whole process when a thread is waiting for I/O operations to

complete. Another solution is to have remote calls that do not return a result. But this is not sufficient, if a later synchronization is necessary or if results should be returned. A simple and effective solution is to split a RPC on the client side in two parts: the invocation of the procedure and the claiming of the results. This concept is called *asynchronous RPC*. It allows parallel programming by issuing several remote calls (possibly to different servers) and collecting their results later in an explicit synchronization. From the point of view of control flow, a synchronous RPC is much like a co-routine transfer, while an asynchronous RPC is more similar to spawning a task to do the work and synchronize after its execution is complete. Fig. 2 shows the differences in control flow between a synchronous and an asynchronous RPC. Note that server programs are not affected and need not be modified.

A great advantage of programming with RPCs is that the client and server stubs can be generated automatically on the basis of a *remote interface definition* for each service. A remote interface definition is a specification of the operations to be exported by a service. Thus, the network can be almost completely hidden from the application programmer. There is no need to write network-related code. In most cases, only minor modifications are necessary to turn a sequential application into a distributed one. Furthermore, the interface definition itself is a good documentation of the functionality provided by a service.

3 Software Overview

The SCIDDLE software package is composed of the following three parts,

- the stub generator,
- the user library/runtime system,
- the super-server.

The *stub generator* (or stub compiler) generates the client and server stubs on the basis of a concise specification of the server interface, the *remote interface definition*. A remote interface definition module is needed for each service to be used by the application. This module comprises a declaration of all procedures exported by the particular service as well as the types of their parameters, exception conditions and possibly context variables. This specification defines the application-level communication protocol between client and server. The interface definition language and the stub compiler are described in section 4.

The client program contains, beside RPCs, a couple of calls to the SCIDDLE *user library*. The user library provides for service initialization, server startup and termination facilities, synchronization of asynchronous RPCs and easy management of on-going asynchronous RPCs. The client usually starts up a *pool* of servers for each service to which it later issues remote calls. After completing its work the client closes all the services, which causes all running servers to be terminated. The user library interface will be presented in more detail in section 5.

The *runtime system (RTS)* includes a message subsystem to support message passing between client and server stubs. It is responsible for efficient and reliable delivery of messages and for data format conversion if required. The RPC protocol is built on top of the message system. The RTS is able to detect machine, process or communication failures.

The *super-server* is a daemon process which resides on all server machines. In cooperation with the corresponding client parts in the user library it starts and terminates server processes, delivers signals and it is responsible for user authentication on the remote machines. The runtime system and the super-server will be discussed in section 6.

4 The Remote Interface Definition Language

The SCIDDLE remote interface definition language is a declarative language that serves to specify the set of context variables and procedures exported by a service. Constants and types may be defined for the context variables and procedure parameters.

4.1 Basic Language Elements

The two essential language constructs are *type* and *procedure* declarations. The following basic data types are supported: short and long integers, single and double precision floating point numbers, single and double precision complex numbers, characters and strings. One- and multi-dimensional arrays and records may be constructed from these basic types. User-defined types allow to build arbitrary data types within this type system. *Constants* may be defined and used as symbolic array dimensions or for other purposes. Array handling will be discussed in detail in the following subsection.

Procedures come in two flavors: synchronous and asynchronous. Each procedure parameter is tagged with a direction attribute (IN, OUT, or INOUT). Parameters are passed *by-value-result*, which means that they are copied in the respective directions.

The following example illustrates, how the interface definition of a service for the multiplication of complex matrices might look like:

```
INTERFACE Example;

CONST
    MaxRow = 100;
    MaxCol = 100;

TYPE
    Matrix = STATIC ARRAY [MaxRow, MaxCol] OF COMPLEX;

PROCEDURE MatMult(IN a, b: Matrix; OUT c: Matrix): SYNC;
(*
 * computes complex matrix multiplication c = a * b
 *)

END
```

Moreover, *context variables* can be defined. Context variables provide a means for transferring data to the static (global) data area of a server. Context variables are global variables of the server and thus persist across remote calls. Context variables allow the client to update and read server context. This is done by passing context parameters to remote procedures. These special parameters are not passed to the server procedure, but modify or return the values of a server context variable depending on their direction attribute. With the introduction of context variables we gain some efficiency as we can avoid to copy (local) parameter data to global variables, when static data is needed. But, first of all, we save memory space. This is crucial in the application we discuss in section 8, where constants and geometry defining variables make up most of the (huge) memory requirements.

4.2 Array Handling

As SCIDDLE is designed in particular for parallel distributed numerical applications, it provides for special array handling support for easy distribution of subarrays to multiple servers. A subarray can be selected by attributing array parameters with *views*. Views are, like array sections in Matlab [18], a collection of triples of the form <base-index: stride: end-index>. The view components are either constants or the names of other (integer) parameters passed in the same call. As an example, let us rewrite the above procedure declaration for a parallel implementation of the matrix multiplication algorithm. Let us assume that we cut the matrix *a* into horizontal stripes and give each of the compute servers one such stripe. It will eventually return the corresponding stripe of the result matrix *c*. The new procedure declaration could look like

```
PROCEDURE ParMatMult(IN lo, hi          : INTEGER;
                     IN  a[lo:hi,:], b : Matrix;
                     OUT c[lo:hi,:]    : Matrix): ASYNC;
```

Note that the procedure is now declared to be asynchronous, because we want the servers to compute in parallel. `lo` and `hi` are the base and end index, respectively, of the stripe passed in an actual call.

Figure 3: (a) unpacked array with view, (b) same array packed

Array types are attributed with a *storage class* and a *packing specification*. The storage class can be specified as either static or dynamic. Static arrays have a size fixed at compile time and are allocated in the global area or on the stack, whereas dynamic arrays have no predefined size and are allocated on the heap. Packing refers to an index space transformation. Packed means that the index space is normalized to start with one and have stride one in all dimensions, i.e. the array elements are packed together and shifted towards the index 'zero' point.

In order to allow efficient memory allocation for subarrays on the server side, arrays should be declared dynamic and packed. If the array is dynamic and packed, memory is allocated only for the number of occupied array elements instead of the whole array. Fig. 3 illustrates this for a sample 4-by-5 matrix and the view [2:2:4, 1:2:5]. However, the programmer must take this transformation into account in the server program.

4.3 Exceptions

An *exception* is defined as an application-level failure of a remote procedure. A local procedure would return an error code as an indication that the normal results could not be returned. With remote procedures we do not want to send back useless return values in this case. Rather we would like to return a simple exception code indicating the type of failure. This is done by listing symbolic names for the possible exceptions for each procedure. For example, a numeric algorithm might be considered failed, if there is no convergence after a number of iterations, if a matrix is badly conditioned or if an arithmetic exception (such as division by zero or underflow/overflow) occurs. The definition of such a procedure with exceptions could have the following appearance:

```
PROCEDURE Solve(IN a: Matrix; IN b: Vector; OUT c: Vector);
EXCEPTION NoConvergence, BadCondition, ArithmeticException;
```

The declaration of exceptions is simply mapped to constants. When an exceptional condition is detected by the application, the library routine `sc_exception()` is called with the corresponding exception code as an argument. This causes the server procedure to be quit and the exception code to be returned to the client. The result value of the RPC will be a positive exception code instead of zero on success (or a negative RTS error code).

4.4 The Stub Generator

The stub generator takes a remote interface definition as input and produces the client and server stub modules, a header file for each stub module and a common header file. The client stub contains the stub routines for the client side, one for each synchronous and two for each asynchronous remote procedure. They appear to the client as if they were the server procedures, except for a prefix and an additional control parameter. The prefix is necessary to distinguish recursive calls to the server procedure (local call) from nested calls to the same procedure on a different server (remote call). For example, the above synchronous matrix multiplication procedure would produce a client stub procedure with the following interface:

```
int call_MatMult(Matrix a, Matrix b, Matrix c, int sid);
```

The additional last parameter identifies the invoked server. The procedure returns a status code. The type `Matrix` is defined in the header file. The asynchronous version of this example generates an invoke and a claim procedure with these interfaces:

```
int invoke_ParMatMult(int lo, int hi, Matrix a, Matrix b, int sid);
int claim_ParMatMult(Matrix c, int cid);
```

The parameter list of the invoke procedure contains all IN and INOUT parameters of `ParMatMult` and, additionally, the identifier of the called server. It sends an invocation request to the server and produces a *call identifier*. On the other hand, parameters declared INOUT and OUT appear in the parameter list of the claim counterpart. This procedure consumes a call identifier and blocks the caller until the results arrive. It returns a status code. The purpose of call identifiers is to associate results with invocations. A claim stub will return only those results corresponding to the passed call identifier. Calling it with an identifier not produced by a previous call to the associated invoke stub, will return an error. Call identifiers are also very useful for managing a set of on-going calls as we will see in section 5.2.

The status code returned by the call and claim stubs indicates the type of return as well as a return code itself. Three return classes can be distinguished:

1. *Normal return.* The procedure has completed successfully and the normal result values could be read into the result parameters. This condition is indicated by a zero status code.
2. *Error return.* An error has occurred in the client or server stub. A negative status code denotes the error condition. The results may have been read or not.
3. *Exception.* An exception was raised in the server procedure. No results have been transferred back to the client. The status code contains a positive exception code.

The server stub on the other hand contains the server stub procedures which receive arguments, call the actual server procedures and return results. Moreover, it contains the two procedures `setup_<iface-name>()` and `run_<iface-name>()`, where `<iface-name>` is the name of the interface definition, i.e. the identifier appearing after the keyword `INTERFACE`. The former initializes the server process and registers the server stub procedures with the dispatcher table. In the latter a connection from the client is accepted and the call dispatcher is entered, which repeatedly receives requests and calls the appropriate server stub procedure. The call dispatcher never returns. Both of these routines must be called from the user-supplied server main program. Between the two calls there is space for application initialization.

Figure 4: SCIDDLE application components

Additionally, both stub modules contain routines for sending and receiving user-defined data types, and for allocation and deallocation of these types, if they contain dynamic data structures.

Fig. 4 shows the components of a SCIDDLE application. The client program is composed of the user-supplied client routines, the client stub and the RPC library. The server program consists of the server stub, the user-written server procedures and the RPC library. The latter is linked to both, client and server program.

Language heterogeneity. Stub generator options tell whether the stubs shall interface with user code written either in Fortran 77 or C (K&R or ANSI C or C++). The main differences between Fortran and C which are of interest in this connection are: (a) parameter passing semantics and (b) data structures and their representation. Parameters are all passed by reference in Fortran. Fortran does not support record types. An error message is output when an attempt is made to generate stubs for Fortran from an interface containing record types. Fortran also has different memory representations for multi-dimensional arrays. It stores the array elements in reversed dimension order (column major), while C stores them in actual dimension order (row major). The stub compiler deals with these differences and adapts the emitted stub code to the respective language. SCIDDLE programs written in Fortran need to be linked additionally with the Fortran-to-C *interface conversion library*, which provides the Fortran interface to the user library.

5 The User Library Interface

This section deals with the functionality available to the user through the user library. Two basic blocks of functionality can be distinguished in the user library: service and server management and management of on-going asynchronous calls.

5.1 Service and Server Management

Since SCIDDLE servers are private to their clients, they are started, controlled and terminated by the client. These operations are performed in cooperation with the super-servers residing on each

server machine. Before a server can be started the corresponding service needs to be attached to the client process. Thereby, a symbolic name (a string) is assigned to the service which is then used as a reference to start servers. Servers can be started individually on a specified host or from a configuration file that lists all servers participating at the actual computation. Multiple servers may run on the same host.

During execution a client may want to interrupt a server. It can do so by sending a signal. This can for instance be useful to request status information, receive incremental results on request or to get a progress report.

After the execution is completed the servers should be terminated by the client. A warning message is printed, if there are still on-going calls to servers. Detaching a service is equivalent to terminating all servers providing the service.

5.2 Call Management

Asynchronous RPCs require some means to manage on-going calls. The concept for doing so was taken from Walker et al. [30], who presented an asynchronous RPC facility for the Cronus distributed operating system. Invoking an asynchronous remote procedure produces a call identifier (called a *future* in [30]). The results of a call are uniquely connected to this identifier. It is passed as an argument to the claim procedure. Once the results have been claimed, the call identifier is automatically discarded. Two operations are defined on active calls. These are

```
int sc_callisready(int cid, long timeout);
int sc_calldiscard(int cid);
```

`sc_callisready()` checks the results of a call for availability. It waits for the specified time interval and returns either an indication that the results are available or a timeout occurred. The second procedure, `sc_calldiscard()`, discards the results of a call. After that, the call identifier is invalid.

Parallel programs usually issue several calls to a collection of servers and then wait for the results. In order to allow an easy management of a set of active calls, we allow call identifiers to be included into *call groups*. The whole group can then be tested for readiness returning the identifier of the first available result. The following code fragment illustrates these facilities:

```
/* create an initially empty call group */
cgid = sc_cgcreate();

/* invoke several procedures and add the call ids to the group */
for (i = 0; i < NUMSRVS; i++)
{
    cid = invoke_sampleproc(<arguments>, sid[i]);
    sc_cgadd(cgid, cid);
}

/* claim the results in the order they become ready */
while (sc_cggetsize(cgid) > 0)
{
    cid = sc_cgextractready(cgid);
    claim_sampleproc(<results>, cid);
}

/* destroy the call group */
sc_cgdestroy(cgid);
```

First, an empty call group is created. Second, a number of asynchronous calls to a collection of servers is issued and the produced call identifiers are added to the call group. At this point, the servers are occupied with the execution of the procedure `sampleproc` and the client could perform some other task. Since there is nothing else to do in this example, we call the library procedure `sc_cgextractready()` in order to get the call identifier of the first call to complete. This identifier

is automatically excluded from the call group and consumed in the following call to the claim procedure for `sampleproc`. This process is repeated until we have claimed all results and the call group has no members left.

The operation `sc_cgdiscardall()` discards all call identifiers in the group.

5.3 Miscellaneous Functions

Besides the above two classes, there are a couple of additional functions concerning exceptions and memory management.

A server may raise an exception by calling `sc_exception()` with the exception code as parameter. This causes the current server procedure to be aborted and the exception code to be returned to the client instead of the normal results.

The routine `sc_setbufsize()` sets the send and/or receive buffer sizes of the message subsystem allowing the user to adapt these buffers to the needs of the application. The routines `sc_allocarray()` and `sc_freearray()` allocate and free k-dimensional arrays, respectively.

6 Runtime System Implementation

6.1 Server Process Startup and Termination

Server processes are started and terminated by the SCIDDLE super-server. It receives requests from clients to initiate application servers and to deliver signals to these servers. The process of starting up an application server is depicted in Fig. 5 and can be outlined as follows. First, the client connects to the super-server on the machine where the new server should run. The super-server's port to connect to is read from the host configuration file. The client passes the necessary arguments including the name of the service to be run to the super-server (step 1). Second, the super-server looks up the name of the executable for the service name in the SCIDDLE services file, creates a port for the new server and starts it up (step 2). Third, on success, it sends the port number or, on failure, an error code back to the client (step 3) such that the client knows the address (port) where the new server can be contacted (step 4).

Figure 5: Starting up application servers

The super-server keeps track of the processes it started up in a server process table. This is done in order to check possible following requests to deliver a signal for their legitimacy. Signals are only delivered to processes started by the super-server for the respective user. Illegitimate requests are rejected. Servers are terminated by sending them a termination signal. Terminated servers are removed from the internal process table.

The super-server can operate in two modes: single-user and multi-user. In single-user mode it can be started by the user itself and it will accept only requests from that user. In multi-user mode it needs to be run as a root process, but accepts requests from any user. Multiple super-servers can coexist on a machine.

The super-server is also responsible for user authentication. Security is an important issue above all on supercomputers. When started with the respective option the super-server asks for a password before it starts a server. Only services can be provided which appear in a separate SCIDDLE services file.

6.2 Message Subsystem

The message subsystem underlies the RPC protocol. Since we want to solve large-scale problems, the message system is based on the following assumptions: (a) messages are large, (b) available buffer space is limited and possibly smaller than the largest message and (c) messages will be consumed by the application mostly in the order they arrive.

SCIDDLE uses virtual circuit communication based on stream sockets with the TCP/IP protocol suite [24, 25]. Virtual circuits transfer data reliably and are well-suited for bulk data transfer. Since they are connection-oriented, they have the drawback of a time-consuming connection setup, but the amount of data transferred over a once-established connection should outweigh the initial effort. A limitation is that the number of descriptors per process for virtual circuits is quite limited². This restricts the amount of parallelism to that number, but in most cases this should be sufficient. Local communication (within a machine) uses local IPC primitives (UNIX domain sockets).

On top of these communication channels lies the data representation layer. In order to support communication in a heterogeneous environment, data is converted into a standard format before transmission and is converted back to the native format of the receiving host after reception. This process is called *encoding* and *decoding*. We use Sun's XDR (eXternal Data Representation) for the standard format [25]. In order to optimize transfer rates this conversion is switched off between machines of the same architecture.

Because of assumption (b), we can not afford to buffer whole messages in memory. Because of the request-reply nature of RPC, messages are consumed in the order they arrive. This is at least true for synchronous RPC. But also for asynchronous RPC it is most efficient to claim the results of possibly multiple calls in the order they arrive (c). This led us to a buffering mechanism, where messages are sent and received through circular send and receive buffers. They are only put into receive queues, if a different than the actual message is explicitly requested by the application (if so, being aware that this may consume a lot of memory or even burst the memory capacity (b)).

6.3 Failure Detection/Handling and Orphan Treatment

SCIDDLE applications may assume a tree-structured communication pattern. If a process in this tree fails due to a process or machine crash, the whole subtree rooted in this process is considered failed. The client of a failed server detects that the connection to the server is broken. In reaction to this, it calls an error handler for the corresponding service. In most cases, the whole application is considered failed and execution is stopped. So, the default error handler terminates the client, but the user may install his own handler which takes some appropriate actions. For example, a new server could be started and the same task reissued. If, on the other hand, a server detects a client failure, the server will terminate itself.

The drawback of this connection-oriented failure detection is that a failure condition can only be detected, when an attempt is made to communicate with the peer process. But what happens with a server whose client machine has crashed just after calling a procedure? Assume the procedure

²A typical value for workstations is 64.

performs very time-consuming calculations. A server without a client is termed an *orphan*. Orphans potentially waste a large amount of resources. To prevent this an orphan detection and killing mechanism is needed. Since we do not support crash recovery and there is only one client, we can be satisfied with a very simple solution³. Each server process periodically probes whether its client is still alive by checking its connection with the client. If the connection is broken, the client is considered failed and the server terminates itself and all subordinate servers.

7 An illustrative example

In this section we discuss how the computation of the scalar product

$$s = \mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^n a_i b_i \quad (1)$$

of two real vectors $\mathbf{a}$ and $\mathbf{b}$ of length n is distributed on a workstation network with SCIDDLE. This application is of course not suited for a distributed implementation as the communication-to-computation ratio is too high, i.e. is $O(1)$. But it is well suited to demonstrate some of SCIDDLE's features, in particular, the situation where servers are clients to other servers. To get parallelism, the sum in (1) is simply split in m partial sums of equal length which then can be computed in parallel on different processors. (To avoid cumbersome index calculations, we assume that m divides n .) These partial sums can then be computed in parallel on different processors. By continuing this splitting procedure we quickly get a large number of parallel processes. The dependency graph among these processes forms a m -ary tree. Fig. 6 shows an example with $m = 4$ and $p = 1$. There are two procedures involved in the computation, `main` and `scalar`. `main` just sets up the computation. It calls the server at the root of the tree, an instance of `scalar`. (Notice that we do not consider the main program (master or top most client) but the first server to be the root of the tree.) `scalar` computes (1) if $p = 0$ or spawns m further servers of its own to compute partial sums.

Figure 6: Tree of degree $m = 4$ and of height $p = 1$.

This algorithm has been implemented in C. (For the Fortran analogue see Appendix A.) The interface between client and server is defined in the file `scalar.sc` which does not differ for C and Fortran applications.

³In contrast, the RPC system Radjoot [20] provides a sophisticated orphan treatment mechanism supporting crash recovery and multiple clients


```

INTERFACE Scalar;

(*
 * scalar.sc: Interface Definition Module for distributed scalar product
 *)

CONST      LDQ = 1000;

TYPE      Vector = PACKED DYNAMIC ARRAY [LDQ] OF LONGREAL;

PROCEDURE scalar (IN  myid, n:      INTEGER;    (* node id, vector length *)
                  IN  m, p:      INTEGER;    (* degree, height of tree *)
                  IN  a[n], b[n]: Vector;    (* a, b vectors *)
                  OUT s:          LONGREAL    (* scalar product a*b *)
                  ): ASYNC;

END

```

The client's main program is given below. Notice that the physical length **LDQ** of the two vectors is defined in the stubs via the interface. However, only n elements are transmitted in each call as both vectors are associated with a corresponding view.

```

/*****
 *   main.c: client main program for distributed scalar product
 *****/

#include <stdio.h>
#include "Scalar_clt.h"

#define NICEVALUE      0

main(argc, argv)
int argc; char **argv;
{
    double d[LDQ], e[LDQ], s;
    int i, m, n, p, cid, sid;

    /* input  n,  p,  m */
    . . .

    for (i = 0; i < n; i++) {
        d[i] = 1.0; e[i] = i;
    }

    /* init Sciddle master */
    sc_initmaster(&argc, argv);

    /* attach service and start first server on localhost */
    sc_svcattach("SCALAR", init_Scalar);
    sid = sc_srvstart("localhost", "SCALAR", NICEVALUE);

    /* invoke SCALAR server at the root of the tree */
    cid = invoke_scalar(0, n, m, p, d, e, sid);

    /* claim the result */
    claim_scalar(&s, cid);

    /* terminate server and detach service*/
    sc_srvterm(sid);
    sc_svcdetach("SCALAR");
}

```

```

    printf(" s = %10.2f\n\nEnd of program.\n",s);
}

```

The asynchronous call of procedure **scalar** is embedded into a sequence of calls to the SCIDDLE user library. These calls serve for initialization purposes as well as for server startup and termination. **sc_initmaster()** initializes the runtime system for the master client process. In the next step, the service SCALAR is initialized by calling **sc_srvattach()** and passing a string identifier for the service and an initialization routine exported by the client stub. Then we start the first SCALAR server on the local machine by calling **sc_srvstart()** with the service identifier as the second argument. On success we get a server identifier back which is used to call procedures of this server remotely. At the very end, the routines **sc_srvterm()** and **sc_svcdetach()** terminate the SCALAR server and detach the service from the client, respectively.

The procedure **scalar.c** is listed on page 18. It acts as server *and* as client⁴. Therefore, both header files **Scalar_srv.h** and **Scalar_clt.h** which are created by the SCIDDLE stub compiler have to be included. The array **hosttab** contains the host names of the available compute servers. If the input parameter *p* is zero, **scalar** actually computes the scalar product of the input vectors **a** and **b**. If *p* is positive, **scalar** passes on the work to its *m* server processes. In this case **scalar** plays the role of a client. As the number of the involved servers is unknown at compile time, we use a call group to manage them. The call group is created by **sc_cgcreate()**. It is empty initially. After having started the service SCALAR on a number of processors, the server routine **scalar** is invoked *m* times. Their call id's are added to the call group. Afterwards, the results are claimed as they come in. As long as the cardinality of the call group is positive, **sc_cgextractready()** yields the call id of an earlier called server for consumption of the scalar claim stub. At the same time the call id is removed from the call group. Instead of working with a call group we could make **sid** a vector and claim the results in the order of the invocation of **scalar**. This may lead to idle times on the client if those servers which were invoked later respond first.

The main program of the server looks as follows

```

void main()
{
    setup_Scalar();
    run_Scalar();
}

```

It has to be provided by the programmer! This particular main program could of course have been constructed by the SCIDDLE stub generator and put into the file which contains the server stub. To have the main program available explicitly provides a reasonable place where global variables can be initialized.

8 A large scale computation from quantum chemistry

Among the methods to solve the Schrödinger equation, the Hartree-Fock method is the most important one. The Hartree-Fock equation is formulated as a general eigenvalue problem

$$F\psi = E S \psi, \quad (2)$$

where the Fock operator *F* corresponds to the Hamiltonian and is expressed in a basis of (non-orthogonal) Gaussian type functions. The equation is solved variationally to obtain an optimum energy *E* and the corresponding eigenvectors (electronic charge density) of the Fock equation (2).

The Hartree-Fock equation, even though only based on an approximate theory, gives quite accurate results and is widely used in computational chemistry. Molecular geometries, for example, can be computed with less than 2% error in most cases. More importantly, the Hartree-Fock method serves as a starting point for most higher level of theory methods.

⁴If **scalar.c** were restricted to its server function, only the else-part of the if-statement would appear and only the parameters *n*, *a*, *b*, *s* and the local variable *i* were necessary.

```

/*****
*   scalar.c: demo procedure for distributed scalar product.
*****/

#include "Scalar_srv.h"
#include "Scalar_clt.h"

#define NICEVALUE    0

static char *hosttab[] =
{
    "ru1.inf.ethz.ch", "ru2.inf.ethz.ch", "ru3.inf.ethz.ch",
    "ru4.inf.ethz.ch", "ru5.inf.ethz.ch", "ru6.inf.ethz.ch",
    "ru7.inf.ethz.ch", "ru10.inf.ethz.ch"
};

static htabsiz = sizeof(hosttab) / sizeof(char *);

void scalar(myid, n, m, p, a, b, s)
    int    myid, m, n, p;
    double *a, *b, *s;
{
    int    i, k, grpid, sid, cid, serverid;
    double spart;

    if (p > 0) {
        sc_svcattach("SCALAR", init_Scalar);

        /* create a call group */
        grpid = sc_cgcreate();

        k = n / m;

        /* distribute work */
        for (i = 0; i < m; i++) {
            serverid = m*myid+1+i;
            sid = sc_srvstart(hosttab[serverid % htabsiz], "SCALAR", NICEVALUE);
            cid = invoke_scalar(serverid, k, m, p-1, &a[i*k], &b[i*k], sid);
            sc_cgadd(grpid, cid);
        }

        /* claim partial sums in the order they arrive */
        *s = 0.0;
        while (sc_cggetsize(grpid) > 0) {
            cid = sc_cgextractready(grpid);
            claim_scalar(&spart, cid);
            *s += spart;
        }
        sc_svcdetach("SCALAR");
    }
    else {
        *s = 0.0;
        for (i = 0; i < n; i++) {
            *s += a[i]*b[i];
        }
    }
}

```

The method scales with $O(n^3)$ in the number n of basis functions (which is proportional to the number of electrons in the system considered). A geometry optimization of a large molecule (50 to 100 atoms) easily takes several hours of supercomputer time.

The ‘Direct Self Consistent Field Optimization method’ (‘direct SCF method’) to solve the Hartree-Fock equations was first implemented by Almlöf et al. [2] in the program DISCO. The ‘direct’ algorithm bypasses the Input/Output bottleneck of the traditional implementations of the Hartree-Fock model by computing all data required to set up the secular equations as they are needed, rather than to retrieve them from intermediate storage.

This CPU bound algorithm allows much larger problems to be solved than conventional methods could do. The most prominent example of an extremely large application is the series of computations that led to the prediction that ‘Buckminsterfullerene’ or ‘Buckyball’ is a stable molecule before it was found experimentally [15]. Today virtually all Hartree-Fock and Møller-Plesset theory based program packages such as GAUSSIAN90, CADPAC or GAMESS use the direct SCF algorithm.

The computational load in a Hartree-Fock calculation is in the evaluation of the Fock matrix rather than in the solution of the eigenvalue problem (2). However, the computation of the Fock matrix can be split into tasks that can be processed independently and in arbitrary order. On shared memory multiprocessor vector computers such as the Cray Y-MP/8, performances beyond 1 Gflops were obtained rather easily [17] using a parallelized version of the DISCO program [1].

Since the computation to communication ratio in a Hartree-Fock computation scales as n^3/n^2 or n^3/n depending on the type of application (calculation of the molecular energy; calculation of the derivative of the energy with respect to the molecular geometry), this type of coarse grain parallel scheme can readily be extended to a network of (multiprocessor) machines [16]. Using SCIDDLE-DISCO we were able to obtain performances of more than 16 Gflops on a network of two Cray C90 computers in dedicated mode. The two 16 processor machines involved were located at two different centers, the Pittsburgh Supercomputer Center (PSC) and the Cray Research Inc. Corporate Computing Center (CCN) at Eagan, MN, and were connected via Internet. At the time of the experiment (Saturday, August 14, 1993) the average sustained communication rate was approximately 50 kBytes/s. The degree of parallelism obtained for the complete application (including communication) was 99.3%, indicating a raw performance of 29.5 (out of 32) C90 or more than 75 Cray Y-MP processors [21].

Table 1: Network and cooperative computing experiments using the SCIDDLE-DISCO system.

Machines	Sites	Number procs	Communication	Performance [Gflops]	Date	Ref.
Cray C90 Cray C90 (dedicated)	CCN PSC	32	Internet ~ 50 kBytes/s	> 16	Aug 93	[21]
27 IBM RS6000/550 (non-dedicated)	IBM	27 (18)	Ethernet	> 0.5	Aug 92	[29]
Cray Y-MP/2 Cray-2/4 Cray X-MP/4, Cray-2/4 (non-dedicated)	ETHZ ETHL MSC	14 (6)	Internet ~ 35 kBytes/s (Trans-Atlantic)	1.2	Sept 91	[16]
5 Cray Y-MP (dedicated)	CCN	20	FDDI	3.2	Aug 91	[3]
Cray Y-MP/4 (Client) 20 RS 6000/590	ETHZ ETHZ	(15)	Ethernet/ FDDI	> 1	Jan 94	

Even higher degrees of parallelism were obtained with the IBM T. J. Watson Research Center (Yorktown Heights, NY) ‘WATAIX’ cluster of workstations consisting of 27 RS6000 (models 540, 550 and 560; Ethernet connected). We were able to obtain performances between .5 to .75 GFlops even though the cluster was in non-dedicated mode. On the average each node was available approximately 50% to our application. A prerequisite to obtain such performances was

to successfully address the task-scheduling problem at the application level. Using the scheduler described in [29] we obtained a degree of parallelization of 99.5% fairly consistently.

Already in 1991 we were able to obtain GFlops performances on local and wide area networks of Cray supercomputers [3, 16]. Perhaps the most impressive result was a 1.2 GFlops performance recorded for a “Trans-Atlantic Network” consisting of 2 Cray-2 and 2 Cray X-MP, a total of 14 processors based in Switzerland at the ETH Zürich (ETHZ) and ETH Lausanne (ETHL) computer centers and the Minnesota Supercomputer Center (MSC).

A summary of the various networking experiments is given in Table 1. In this table the number of processors given is the sum of processors of all machines in the network or cluster. The number in parenthesis indicates the processor number actually available in the computation. The performance measurements are based on the operation count of the Cray version of the program.

More recently, using SCIDDLE 3.0, we started to link heterogeneous types of computers. There are two major incentives to proceed in this direction. On one hand, as demonstrated e.g. by our WATAIX implementation, networking high-performance workstations of a domain, e.g. a department, very often yields turnaround times in computations that challenge supercomputer performances. As such clusters quite often link machines with different number representations, a heterogeneous SCIDDLE implementation was necessary to exploit the full power of such networks. Combining three Silicon Graphics Indigo-2 and one IBM RS6000-530 workstation of our own laboratory, we were able to obtain about 20% of the performance of a single Cray Y-MP processor. Whereas the task-scheduling on this kind of network does not pose any problems, the application code optimization (adjustment of algorithms, etc.) for each machine involved often represents a major effort.

On the other hand, applications may be partitioned into tasks with very different resource or service requirements. These may be as diverse as data retrieval, vector/parallel compute or graphics services. Such various services can be made conveniently available only in a heterogeneous network of computers. A heterogeneous network environment as SCIDDLE enables scientists to start and monitor their entire application from a single window.

On a homogeneous cluster of computers the sequential part in DISCO takes about 0.5% of the overall execution time [29]. If this part is run on a faster computer than the parallel part the sequential fraction of the computation is reduced. Notice that by a reduction of the sequential part from 0.5% to 0.25% Amdahl’s law predicts a speedup of 67 vs. 80 on a cluster of 100 processors.⁵ Running the SCIDDLE-DISCO system on a cluster consisting of 20 IBM RS-6000/590 and a 4-processor Cray Y-MP, which, as the client, handled the serial part of the computation an increase of parallelism of the above amount consistently was observed. Replacing the Cray by the NEC SX-3 at the Swiss Supercomputer Center we could have reduced the turn-around time even more.

9 Concluding remarks

We presented SCIDDLE, a portable, easily usable communication environment for distributing parallel applications on heterogeneous networks of workstations and supercomputers. SCIDDLE is based on the client-server programming model. This model allows the introduction of parallelism by means of asynchronous remote procedure calls. The user defines the interface between communicating processes in a simple declarative language. The functionality of SCIDDLE such as starting of remote processes, synchronization, etc., is provided through calls to subroutines of the user library.

As SCIDDLE supports the client-server model, the possible communication patterns have the structure of trees. This is the communication pattern of embarrassingly parallel applications, which are those applications which have been implemented with highest efficiency on computer networks. The performance gained for the quantum chemistry application discussed in the previous section, an embarrassingly parallel application as well, exceeded the peak performance of the fastest supercomputers available today.

Working with remote procedure calls has several advantages. First, the communication is provided through subroutine calls, a well understood mechanism for transferring data and control

⁵The asymptotic limit of the speedup is increased from 200 to 400.

from a program unit to another. Second, because of its simplicity the communication code between processes can be generated automatically (stubs). Third, the code that is going to run in parallel has to be changed only in a few places. This guarantees an easy incorporation into existing programs. In the quantum chemistry application that we discussed in the previous section only the client routine had to be changed. (There are a few header files which have to be included by the server program.)

In most cases we used multiprocessor supercomputers for our application runs for this is the most convenient way to get very high compute power. This also means that we get many processors with relatively few machine accesses and relatively few data transfers over the slow networks.

When working with workstations we have to use large numbers of them to get real high performance. This means that we have to prepare runs, i.e. copy and eventually compile files on quite a few machines. Management of parallel applications on large workstation clusters thus becomes a major issue. We will address this problem in subsequent work.

Acknowledgments

This work was supported in part by a University Research and Development Grant of Cray Research Inc., by the NEC Fund of the Swiss Scientific Computing Center, Manno, and by a grant from IBM Switzerland. The distributed computations have involved the following computer centers: ETH Zürich, ETH Lausanne, MSC Minneapolis, and the IBM T. J. Watson Research Center. Generous grants of computer time, as well as excellent technical support from personnel at these centers have been instrumental for the successful completion of this project. The authors would also like to thank Cray Research and the Pittsburgh Supercomputer Center for providing access to a network of dedicated machines.

References

- [1] J. Almlöf, K. Faegri, Jr., M. W. Feyereisen, T. H. Fischer, K. Korsell, and H. P. Lüthi. DISCO, a direct SCF and MP2 program. For reference see e.g. [2] or [17].
- [2] J. Almlöf, K. Faegri, Jr., and K. Korsell. Principles for a direct SCF approach to LCAO-MO ab-initio calculations. *Journal of Computational Chemistry*, 3:385–399, 1982.
- [3] P. Arbenz, H. P. Lüthi, J. E. Mertz, and W. Scott. Applied distributed supercomputing in homogeneous networks. *International Journal of High Speed Computing*, 4:87–108, 1992.
- [4] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2:39–59, 1984.
- [5] J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, New York, 1987.
- [6] R. Butler and E. Lusk. *User's Guide to the P4 Programming System*. Argonne National Laboratory, October 1992.
- [7] C. H. Cap and V. Strumpfen. The Parform - a high performance platform for parallel computation in a distributed workstation environment. Tech. Report 92.07, Institut für Informatik der Universität Zürich, Zürich, Switzerland, June 1992.
- [8] C. H. Cap and V. Strumpfen. Efficient parallel computing in distributed workstation environments. *Parallel Computing*, 19:1221–1234, 1993.
- [9] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32:444–458, 1989.
- [10] B. Chapman, P. Mehrotra, and H. Zima. Vienna Fortran — A Fortran language extension for distributed memory multiprocessors. ICASE Report 91-72, NASA Langley Research Center, Hampton, VA, September 1991.

- [11] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Tech. Report COMP TR90079, Department of Computer Science, Rice University, Houston, TX, March 1991.
- [12] A. Goscinski. *Distributed Operating Systems: The Logical Design*. Addison-Wesley, 1991.
- [13] R. Hempel. *The ANL/GMD Macros (PARMACS) in FORTRAN for Portable Parallel Programming using the Message Passing Programming Model, User's Guide and Reference Manual*. Gesellschaft für Mathematik und Datenverarbeitung mbH, November 1991.
- [14] B. W. Lampson. Remote procedure calls. In B. W. Lampson, M. Paul, and H. J. Siebert, editors, *Distributed Systems - Architecture and Implementation*, pages 365–370. Springer-Verlag, Berlin, 1981. (Lecture Notes in Computer Science, 105).
- [15] H. P. Lüthi and J. Almlöf. Ab initio studies of the thermodynamic stability of the icosahedral C₆₀ molecule “Buckminsterfullerene”. *Chemical Physics Letters*, 135:357–360, 1987.
- [16] H. P. Lüthi and J. Almlöf. Network supercomputing: A distributed-concurrent direct SCF scheme. *Theoretica Chimica Acta*, 84:443–455, 1993.
- [17] H. P. Lüthi, J. E. Mertz, M. W. Feyereisen, and J. E. Almlöf. A coarse-grain parallel implementation of the direct SCF method. *Journal of Computational Chemistry*, 13:160–164, 1992.
- [18] The MathWorks, Inc., Natick, MA. *MATLAB User's Guide*, August 1992.
- [19] B. J. Nelson. *Remote Procedure Call*. PhD thesis, Computer Science Department, Carnegie Mellon University, May 1981. Published as CMU report CMU-CS-81-119 and Xerox PARC report CSL-81-9.
- [20] F. Panzieri and S. K. Shrivastava. Rajdoot: A remote procedure call mechanism supporting orphan detection and killing. *IEEE Transactions on Software Engineering*, 14:30–37, 1988.
- [21] Pittsburgh Supercomputer Center. *Projects in Supercomputing 1994*. Pittsburgh, PA, 1994.
- [22] A. Z. Spector. Performing remote operations efficiently on a local computer network. *Communications of the ACM*, 25:246–260, 1982.
- [23] Ch. Sprenger. User's guide to SCIDDLE version 3.0. Technical Report 208, ETH Zürich, Computer Science Department, December 1993.
- [24] W. R. Stevens. *UNIX Network Programming*. Prentice Hall, Englewood Cliffs, NJ, 1990.
- [25] Sun Microsystems. *Network Programming, SunOS Manual, Vol. 9*, May 1988.
- [26] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2:315–339, 1990.
- [27] A. S. Tanenbaum and R. van Renesse. A critique of the remote procedure call paradigm. In R. Speth, editor, *Research into Networks and Distributed Applications*, pages 775–783. Elsevier Science Publisher (North-Holland), 1988.
- [28] B. H. Tay and A. L. Ananda. A survey of remote procedure calls. *ACM Operating Systems Review*, 24:68–79, 1990.
- [29] St. Vogel, J. Hutter, T. H. Fischer, and H. P. Lüthi. Direct SCF structure optimization of large molecules on networks of workstations. *International Journal of Quantum Chemistry*, 45:665–678, 1993.
- [30] E. F. Walker, R. Floyd, and P. Neves. Asynchronous remote operation execution in distributed systems. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 253–259, Los Alamitos, CA, 1990. IEEE Computer Society Press.

A Appendix: Fortran program listings for the scalar product example

Here, the main program `main.f`, the server program `scalar.f` and the server main program corresponding to the C listings of section 7 are listed. Notice that in general underscores are replaced by the letter 'f'. The value of the C functions (return code, call id, service id) becomes an output parameter in the Fortran subroutines. It is *appended* to the parameter list in C.

```
c*****
c   main.f: Fortran client main program for distributed scalar product
c*****
```

```
      parameter (LDQ = 1000, NICEVALUE = 0)

      double precision d(LDQ), e(LDQ), s
      integer i, m, n, p, cid, sid, rc

c---- input variables  n, p, m
      . . .

      do 10 i=1,n
         d(i) = 1.0
         e(i) = dble(i-1)
10    continue

c---- init Sciddle master
      call scfinitmaster(rc)

c---- attach service and start first server on localhost
      call scfsvcattach('FSCALAR', initfScalar, rc)
      call scfsrvstart('localhost', 'FSCALAR', NICEVALUE, sid)

c---- invoke SCALAR server at the root of the tree
      call invokefscalar(0, n, m, p, d, e, sid, cid)

c---- claim the result
      call claimfscalar(s,cid,rc)

c---- terminate server and detach service
      call scfsrvterm(sid,rc)
      call scfsvcdetach('FSCALAR', rc)

      print *, 's = ', s
      print *, 'End of program.'

      stop
      end
```

```
c*****
c   scalar.f: Fortran demo procedure for distributed scalar product.
c*****
```

```
      subroutine scalar (myid, n, m, p, a, b, s)
      parameter (NICEVALUE = 0)

      integer myid, m, n
      double precision a(n), b(n), s

      integer i, k, p, grpId, sid, cid, serverid, htabsize, rc
```



```

double precision spart

character*(16) hosttab(8)
data hosttab /'ru1','ru2','ru3','ru4','ru5','ru6','ru9','ru10'/
htabsiz = 8

external initfScalar

if (p .gt. 0) then
    call scfsvattach('FSCALAR', initfScalar, rc)

c----- creat a call group
call scfcgcreate(grpid)

    k = n / m

    do 10 i = 1, m

c----- start servers
        serverid = m*myid+i
        call scfsrvstart(hosttab(mod(serverid,htabsiz)+1),
>            'FSCALAR',NICEVALUE,sid)

c----- invoke the SCALAR servers
        call invokefscalar(serverid, k, m, p-1, a((i-1)*k+1),
>            b((i-1)*k+1), sid, cid)
        call scfcgadd(grpid,cid,rc)
10    continue

c----- claim partial sums in the order they arrive
    s = 0.0
20    continue
        call scfcgextractready(grpid, cid)
        call claimfscalar(spart,cid,rc)
        s = s + spart
        call scfcggetsize(grpid,rc)
        if (rc .gt. 0) goto 20

        call scfsvdetach('FSCALAR', rc)

    else
        s = 0.0
        do 30 i = 0, n
            s = s + a(i)*b(i)
30        continue
    endif

return
end

program main

integer rc

call setupfScalar(rc)
call runfScalar()

stop
end

```