Anonymous Remote Computing: A Paradigm for Parallel Programming on Interconnected Workstations

Rushikesh K. Joshi and D. Janaki Ram, Member, IEEE

Abstract—Parallel computing on interconnected workstations is becoming a viable and attractive proposition due to the rapid growth in speeds of interconnection networks and processors. In the case of workstation clusters, there is always a considerable amount of unused computing capacity available in the network. However, heterogeneity in architectures and operating systems, load variations on machines, variations in machine availability, and failure susceptibility of networks and workstations complicate the situation for the programmer. In this context, new programming paradigms that reduce the burden involved in programming for distribution, load adaptability, heterogeneity, and fault tolerance gain importance. This paper identifies the issues involved in parallel computing on a network of workstations. The Anonymous Remote Computing (ARC) paradigm is proposed to address the issues specific to parallel programming on workstation systems. ARC differs from the conventional communicating process model by treating a program as one single entity consisting of several loosely coupled remote instruction blocks instead of treating it as a collection of processes. The ARC approach results in distribution transparency and heterogeneity transparency. At the same time, it provides fault tolerance and load adaptability to parallel programs on workstations. ARC is developed in a two-tiered architecture consisting of high level language constructs and low level ARC primitives. The paper describes an implementation of the ARC kernel supporting ARC primitives.

Index Terms—Anonymous remote computing, cluster computing, remote instruction block, parallel programming.

1 INTRODUCTION

Workstation clusters are becoming increasingly popular in academic and scientific computing. The processing power of the workstation has witnessed tremendous growth resulting in clusters of workstations possessing the computing power of a supercomputer. Many approaches have been proposed for improving the utility of workstation clusters in recent times, the notable ones being the Berkeley NOW project [3] and the Batrun project at IOWA [41]. Broadly, these approaches can be classified into three levels—the operating system level, the resource level and the programming language level. Some examples of operating system level approaches are the Amoeba distributed operating system [30], the V Distributed System [15], the Coda distributed file system [33], Sun’s NFS support, the load leveler on IBM RS/6000 system, and the Batrun’s distributed processing system (DPS). Cooperative file caching [16] and RAID systems [3] are examples of resource level approach. At the programming language level, many programming languages and paradigms have been proposed for loosely coupled distributed systems from the point of view of writing distributed programs as well as developing client-server applications [4], [6]. Parallel programming on workstation systems is a relatively new field and has been an attractive proposition ever since it was used in some of its early forms such as the RPC thread based system of Ball, Renesse, and Tanenbaum [8]. Other examples of parallel programming approaches on workstation systems are the IPC [38] based message passing systems, RPC thread-based systems [32], distributed shared memories [39] such as Mether system [28], programming tools such as PVM [40] and P4 [11] and programming languages such as Orca [7] and Linda [12].

Most of the existing parallel programming approaches on workstation systems follow the collection of processes (COP) model in which processes communicate via message abstractions or shared memory abstractions. Though this model has been widely used in the context of tightly coupled systems such as multiprocessors, it has several drawbacks when applied to loosely coupled openended workstations. Each workstation typically runs its own operating system and provides a multiuser environment. Workstations in a network can be heterogeneous. Loads on workstations keep changing from time to time. Moreover, it may not be possible to support a common file system for all workstations in a domain. Network links and workstations themselves are prone to failures. The number of workstations in a domain keeps changing. These factors pose serious difficulties from the point of view of openendedness, fault tolerance, load adaptability and programming ease, when the COP model is adopted. For example, as the load on the workstations keeps changing dynamically, it is not possible for the programmer to predict the lightly loaded nodes and start the processes on these nodes. Dynamic load variations may slow down the performance of a process...
that has been mapped onto a particular workstation. Similarly, when a node fails during the execution of a program, a process’s state is lost, which requires re-execution of the entire program. Workstations may get connected to a domain during the execution of a program which may not be able to utilize the additional computing power. Such situations make parallel programming on workstation clusters a complicated task. To reduce the burden of programming on these systems, new paradigms are required.

In this paper, we present the Anonymous Remote Computing (ARC) approach to parallel programming on workstation systems that attempts to address these issues at the language level. The ARC computational model takes a diversion from the conventional COP model. A program is treated as one single entity made up of multiple loosely coupled remote instruction blocks. Remote Instruction Blocks (RIBs) are open program fragments which are heterogeneous and fault tolerant. ARC makes distribution and heterogeneity transparent to the programmer. An ARC task (RIB) may be sent by the ARC system for remote execution, or executed on the host node itself if a suitable remote node is not available. An RIB is open in the sense that it may migrate for execution to an anonymous workstation of a different architecture.

ARC provides a two-tiered architecture. At the lower layer, the primitives which are built over a kernel provide the basic support for anonymous remote computing while at the upper layer, various easy-to-use high level programming language constructs are supported. The ARC kernel provides support for migration, fault tolerance, heterogeneity, load adaptability, and asynchronous intimations for availability, keeping the programming layers very elegant. The ARC model has been implemented on a network of workstations consisting of Sun 3s, Sun Sparcs, and IBM RS/6000 workstations. The network has two independent file systems. The RS/6000s are connected via a 100 MB/sec FDDI network and the Suns are connected via a 10 MB/sec Ethernet.

The rest of the paper is organized as follows. Section 2 discusses the issues involved in parallel programming on workstation clusters followed by an overview and analysis of the existing approaches in Section 3. The computational model of ARC is presented in Section 4. Section 5 discusses the two-tiered ARC architecture. In Section 6, the implementation of ARC is presented, followed by the performance study in Section 7.

2 Issues in Parallel Computing on Interconnected Workstations

Parallel computing on tightly coupled distributed systems has so far been widely popular. However, recent advances in communication technology and processor technology make parallel programming on loosely coupled distributed systems, especially on interconnected workstations, a viable and attractive proposition. There are several key issues that distinguish parallel computing on workstation clusters from that of tightly coupled massively parallel architectures. We discuss these issues in detail below in the following four major categories:

1) Changing loads on the nodes of the network
2) Changing node availability on the network
3) Differences in processor speeds and network speeds
4) Heterogeneity in architecture and operating systems.

2.1 Changing Loads on the Nodes of the Network

A distributed system consisting of interconnected workstations experiences a wide fluctuation of loads on individual nodes. There is always a considerable amount of unused computing capacity present in the network. Fig. 1 shows a trace of average loads on four workstations that we obtained in our environment which is predominantly academic. Vanavil and green are IBM RS/6000 workstations. Vanavil is a busy openly used machine with 48 user terminals, whereas green is managed by a load leveler that maintains a steady flow of batch jobs for the consumption of the machine. Bronto and elasmo are two Sun Sparcs which are situated in a smaller laboratory. It can be observed from the figure that the load pattern varies drastically from time to time depending upon the specific user characteristics. It is a challenging task to execute parallel programs in such an environment while achieving a fair amount of load balance. A program is said to be load adaptive if it adapts to the changing load in the system. However a programmer may not be able to use the knowledge of load fluctuation in the program unless an adequate language support is provided.

![Fig. 1. Load variation on four machines in an academic environment.](image-url)
2.2 Changing Node Availability on the Network

Distributed systems are characterized by nodes that keep going down and coming up over a period of time. Ideally, the computation should not get affected by the changing node availability in the network. For example, a program at some given instance of execution may have made use of five nodes and in another instance may have only three nodes available. It is also possible that node failures are transient with respect to one single execution of the program. While computing a subtask, a node or a link might crash, and may again come up before the completion of the execution of the program.

2.3 Differences in Processor Speeds and Network Speeds

Distributed systems consisting of interconnected workstations are characterized by differences in processor speeds and network speeds. The communication overhead in this class of distributed systems is fairly high. Hence these systems are suitable only for coarse grain parallelism. Communication overheads play a major role in influencing speedups of parallel tasks on these systems. For example, an investigation on Apollo Domain network reveals the effect of communication overheads on the speedup for an RPC based system [32]. In a heterogeneous cluster, processors have different speeds. However, during runtime, the effective capacity may vary due to loading effects. Selecting appropriate grain sizes during runtime becomes important as a consequence of this variation. Thus, the programming model aimed at interconnected workstations should provide for easy variation of the task grain size.

2.4 Heterogeneity in Architecture and Operating Systems

Interconnected workstations normally display some amount of heterogeneity in operating systems and architectures. Heterogeneity in operating systems is being bridged through distributed operating systems, distributed file systems, and standardization of software. Distributed file systems such as NFS are common in the interconnected workstation environment to make the file system available over the network. The architectural differences between workstations are more difficult to handle than operating system differences. Since the binary executable files are not compatible between architectures, it will not be possible to execute directly the binaries relating to a task on the other nodes.

The issues discussed above pose several difficulties in parallel programming on workstation clusters. Management of distribution at the programming level further adds to these difficulties. The existing systems address only a subset of these issues. Providing a single approach addressing all the issues, and at the same time keeping parallel systems simple to program and to understand, is a real challenge. Most of the current parallel programming approaches for workstation systems follow the model of collection of processes that communicate via various message passing or shared memory abstractions. The COP model gives rise to many difficulties while programming for variations in loads and speeds, heterogeneity, and fault tolerance. The following section takes an overview of the existing approaches, analyzing their suitability for parallel programming on distributed workstation systems.

3 EXISTING DISTRIBUTED PROGRAMMING APPROACHES

The COP approaches to programming on workstation systems can be grouped under the following categories with some amount of overlap between the categories.

- Bare socket programming
- Remote procedure calls
- Remote execution
- Message passing abstractions
- Shared memory abstractions
- Object-oriented abstractions

Some of them have been proposed for parallel programming whereas others are intended for providing distributed services. We analyze the suitability of these approaches from the parallel programming point of view.

Socket programming. The most primitive constructs for interprocess communication between workstations are sockets. Communication is at the level of untyped byte streams. This approach provides no support for the requirements discussed in almost all the above four categories. Support for intermediate representation such as XDRs is provided in order to handle heterogeneity of data representation. However, selection of nodes, tackling failures, load adaptability, and heterogeneity at the level of executable code have to be handled explicitly by the program.

Remote procedure calls (RPC). RPC attempts to bridge the semantic gap between a local call and a remote call. Communication is typed and the services are predetermined. Usually, parallel programs are written with RPC in combination with light weight threads [8], [32]. There have been attempts to extend the RPC mechanism to heterogeneous environments [10], [42]. These attempts largely focus on the issue of the design of stub generator for heterogeneous RPC systems. However, these constructs do not address the specific issues in parallel programming on loosely coupled distributed systems. RPC requires services to be placed on servers and exported before a program starts executing. A programmer can explicitly start processes on nodes, which might lead to severe load imbalances in the system. In other words, these constructs do not make a distinction between the programmer’s concerns and the system’s concerns in the case of loosely coupled distributed systems. RPC is more appropriate for providing distributed services rather than for writing parallel programs on workstation clusters.

Remote execution. A remote execution facility allows for migrating code fragments to remote servers for execution. The Remote Evaluation (REV) has been developed for providing migration of executable code to servers [37]. The main motivation in developing REV is to reduce the client-server communication. Instead of moving large amounts of data from servers to client for processing, the executable code is migrated to the server site and the results are sent back to the client. A similar approach can also be found in computation...
Remote execution as a standalone paradigm is also not suitable for parallel programming on workstations due to its lack of adequate support for heterogeneity, load adaptability, and fault tolerance.

**Message passing abstractions.** These are the higher level abstractions to bare socket programming. Concept of typed data communication is introduced and untyped byte communication can still be retained. Programming languages such as Lynx [35] and programming platforms such as PVM [40], P4 [11], and MPI [29] support various abstractions of message passing. For example, PVM supports transmission of several data types such as integer or float streams along with byte streams. Some of the problems faced by the bare socket programming model are eliminated here. The model is easier to use due to its elegant abstractions. However, adequate support for heterogeneity, migration, load adaptability and fault tolerance is not provided. If a remote machine fails, the state of a process on that machine is lost and such a failure have to be handled explicitly at the programmer’s level. MPVM [14], a version of PVM, provides support for process migration. However, process migration based solutions are useful for adapting to load variations on homogeneous systems only.

**Distributed shared memory abstractions.** They range from bare distributed shared memory support such as the Mether system [28], TreadMarks [2], and Munin [13] to high level shared memory abstractions such as Orca [7] and Linda [12]. Shared memory abstractions appear to provide much easier programming interface than message passing. Linda supports distributed data structures to which multiple processes can have simultaneous access. Multiple processes can be started on different nodes. In Orca, objects are shared between processes and their children. A new process can be forked explicitly on a processor using a fork construct. Once a process is forked on a machine, it may suffer from loading effects. Once the number of processes is fixed by a program, it cannot utilize the additional computing power that becomes available during runtime.

**Object-oriented abstractions.** In recent years, several object based and object-oriented paradigms such as Emerald [9], Orca [7], CHARM++ [27], and Mentat [18] have been proposed for distributed computing. Processes are replaced by objects. The communication in this case is interobject rather than interprocess. Consequently, the COP model is still preserved but at the level of objects. Various drawbacks of these abstractions in the context of workstation systems have been discussed in [26].

We introduce in this context, the paradigm of **Anonymous Remote Computing** (ARC) that eliminates the problems of the COP model of parallel programming on workstations. Programs are not written as a collection of communicating processes, but as a collection of several loosely coupled Remote Instruction Blocks (RIBs) within a single program entity. RIBs are opened and migrate to convenient anonymous remote nodes during run-time. This approach makes programs fault tolerant, heterogeneous and load adaptable. At the same time, ARC keeps programs easy to understand by separating system’s concerns from programmer’s concerns.

For the programmer, the nodes in the network remain anonymous and a run-time system starts the required processes on remote nodes considering the loads on these nodes. To handle architectural differences, the source code of tasks that are RIBs is identified and stored separately. The mechanisms of migration and lazy compilation of RIBs are used for executing the tasks on heterogeneous nodes. Primitives are provided for handling variations in loads and speeds dynamically.

### 4 The ARC Model of Computation

The ARC model of computation is designed to meet the following two goals of parallel programming on workstation systems:

1) Clear separation between programmer’s concerns and system’s concerns
2) Account for heterogeneity, fault tolerance, load adaptability, and processor availability

The origin of the conventional COP model of parallel programming on workstation systems can be traced to the process based multiprogrammed machine with various interprocess communication mechanisms. Networking of multiple multiprogrammed machines led to a direct adaptation of this model to distributed programming as in SR [5] and Lynx [35], and subsequently to parallel programming as in PVM [40]. The trends in the cluster computers show proliferation of a large number of heterogeneous workstations in a LAN configuration. An important characteristic of this cluster of workstations is dynamically changing loads on them, which makes the task of harnessing the idle computing capacity for parallel programming a very challenging task. The changing load on these systems coupled with link and processor failures make the conventional COP model unsuitable for parallel programming on these systems. The ARC model has been proposed to address these requirements of parallel programming on a cluster of workstations.

In the COP model, since multiple processes are involved, programs are difficult to understand and debug. Explicit programming of communication between entities may lead to synchronization errors. The ARC model aims at eliminating the creation of explicit entities mapped onto different machines. A program in ARC model consists of a single entity comprising of multiple loosely coupled blocks.

Fig. 2 shows an overview of the ARC model. An ARC program consists of several loosely coupled **Remote Instruction Blocks** linked via synchronizers. A remote instruction block is a code fragment that can be executed on an anonymous remote machine. An ARC program may have multiple RIBs that can be executing simultaneously. RIBs are synchronized with the help of synchronizers. RIBs do not involve mechanisms of process creation, or inter-task communication. The nodes on which the RIBs are to execute remain anonymous to the program. The run-time system known as the ARC system decides the nodes on which RIBs are to be executed. The target node thus remains anonymous to the program. It is also possible to migrate an RIB onto an heterogeneous node. An RIB is submitted for a possible remote execution by a user program. At a time, multiple programs can be generating RIBs and multiple
anonymous remote participants can be joining or leaving the ARC system. An RIB can further generate new RIBs.

4.1 Comparing ARC and COP Models

The computing entities in the COP model are processes or objects working directly on top of the operating system, whereas, in the ARC model, they are remote instruction blocks within a single program. Fig. 3 brings out the differences between the current COP approaches and the ARC model based on the patterns of synchronization among the computing entities. The COP approaches are classified into five classes of task synchronous, call synchronous, call asynchronous, message synchronous, and message asynchronous systems.

4.1.1 Task Synchronous Systems

A task synchronous system provides synchronization between computing entities at task level. Two computing entities are synchronized at initiation time or completion time of tasks. The Linda [12] system is an example of task synchronous COP approach. Tasks can be deposited in a tuple space and decoupled processes pickup the deposited tasks.

4.1.2 Call Synchronous Systems

In this system, two computing entities communicate by calling methods defined by each other. When an entity calls a method, it expects an action to be taken and further blocks for the result to return. RPC [31] and the Distributed Processes model [21] are the examples of call synchronous COP systems.

4.1.3 Call Asynchronous Systems

Some of the distributed object-oriented programming systems are call asynchronous systems. In these systems, method calls return immediately in a nonblocking fashion. Either a future mechanism is used or a call back mechanism is implemented to obtain the return value of a call. The latency tolerance mechanism of CHARM++ [27] is an example of a call back system whereas the return-to-future mechanism of Mentat [18] is an example of a future based call asynchronous system.

4.1.4 Message Synchronous Systems

Two computing entities exchange messages such as arbitrary data values. Message communication is blocking. CSP [22] is an example of a message synchronous COP system.

4.1.5 Message Asynchronous Systems

Two entities exchange messages such as data values at any arbitrary point of time. This type of synchronization can be achieved by shared memory as well as message passing techniques. Since a message can arrive at any time, usually messages are typed and the receiving process knows the type of the incoming message. The conventional distributed shared memories (DSMs) and message passing mechanisms such as PVM [40] and MPI [29] are the examples of a message asynchronous COP system.

4.1.6 The ARC Approaches

ARC extends the computing entities in the horizontal domain to RIBs. However, in the vertical domain, ARC can be classified as a task synchronous system. In an ARC system, the synchronization is provided at the beginning or completion of remote instruction blocks that can be seen as loosely coupled tasks. The synchronization is provided with the help of synchronizers as discussed later in this section. Parsets [25] and Object Based Subcontracting (OBS) [26] are two high level ARC approaches proposed by us that fall into this category.

Development of message synchronous and message asynchronous procedural ARC paradigms, and call synchronous and call asynchronous object-oriented ARC paradigms requires the support for communication among remote instruction blocks executing on anonymous nodes. This is a very challenging task and needs further research. The arrowhead in Fig. 3 captures the scope for future developments in ARC paradigms.

4.2 RIBs and Synchronizers

Fig. 4 shows an example adaptive quadrature [4] program depicting the use of remote instruction blocks and the synchronizers. RIBs are the code segments that can be executed at anonymous nodes. Synchronization between independently executing RIBs is achieved through synchronizers.

4.2.1 Remote Instruction Blocks

In Fig. 4, a special variable called RIB variable R is introduced to mark a control specification as a remotely executable block. The variable R is bound to a function call AQ() that computes the area under a curve bounded by given left and right boundaries. Once an RIB variable is bound, a reference to the variable in an EXEC call executes the code associated with the RIB variable. The EXEC call encapsulates system’s concerns such as fault tolerance, heterogeneity and load adaptability from the program. An RIB is migrated to a suitable remote node by the run-time system. In the case of heterogeneity, the RIB source code is migrated and the mechanism of lazy compilation is used to execute the RIB.

The approach of migrating code was exploited earlier by researchers in the case of functional languages. Falcone developed a LISP-based language called NCL to support remote function evaluation (RFE) [19]. In a different approach, Stamos and Gifford developed a remote form of evaluation [37], which introduces the execution of a function at a remote node. In contrast to these approaches, RIBs are openended blocks for which the target machine for execution is unknown until execution time. An RIB execution isolates the programmer from addressing system’s concerns at the programmer’s level.
main () {
    ...
    double left, right, middle;
    double area1, area2, totalArea;
    RIB R;
    SYNC S;
    ...
    middle = (left + right) / 2; // divide the task
    R area1 = AQ (left, middle); // mark one task as RIB
    S = EXEC R; // execute RIB
    area2 = AQ (middle, right); // inlined code of second task
    WAITFOR S; // synchronizer
    totalArea = area1 + area2;
    ...
}

ARC is developed in a two-tiered architecture. At the top layer, RIBs are expressed in a high level program using language extensions to existing languages such as C or C++. At the lower level, the basic primitives for anonymous remote computing are provided. It is possible to develop different higher level paradigms that provide the above two basic characteristics of RIBs. We have earlier proposed two high level ARC paradigms, Parsets and Object Based Subcontracting, which are tailored to the needs of procedural and object-oriented programming, respectively. These two paradigms extract RIBs from specially designed language constructs.

4.2.2 Synchronizers
Synchronizers are mechanisms required to achieve synchronization between independently executing remote instruction blocks. Two types of synchronizers called SYNC variables and lock specifications are provided. Fig. 4 shows the first type of synchronizer, the SYNC variable. An RIB execution is assigned to a SYNC variable S. At any later point, the program can be made to wait on the SYNC variable to assure that the execution sequence of the associated RIB is completed. Functional languages have used similar variables called future variables [20].

Parsets and OBS use lock specifications as synchronizers. In Parsets, function arguments are locked as read-only (RO), write-only (WO), or read-write (RW) during execution. If an argument is RO locked, it can be made available to a subsequent function execution. A similar scheme is used in OBS where locks are obtained on objects for message invocations. Locks provide auto synchronization whereas SYNC variables are explicit synchronizers.

4.3 Dynamically Growing and Shrinking Trees
Computationally, ARC programs can be represented as dynamically growing and shrinking trees as shown in Fig. 5. An RIB can further generate new RIBs. As RIBs complete exe-
cution, the tree shrinks. If more RIBs are generated during execution, a shrinking tree may again start growing. Since ARC is a task synchronous system, synchronization is provided at the beginning or completion of an RIB. Communication between RIBs is not supported. The capability of RIBs to spawn new RIBs along with the task synchronization scheme results in the dynamically growing and shrinking tree model. To achieve different types of RIB communication than just task synchronization is a very challenging task and needs further research.

4.4 Benefits of the ARC Model

The ARC model is based on RIBs contained in a program instead of a collection of processes that explicitly communicate. This approach grants distribution transparency to parallel programs on workstations. RIBs can be made heterogenous by the ARC system by runtime source-code migration and recompilation. In this way, ARC also provide heterogeneity transparency. Transparency to distribution and heterogeneity removes the burden of programming for distribution and handling of heterogeneity at the programmer’s level. The ARC approach grants the property of statelessness to a remote server executing a parallel task. Since ARC avoids the notion of processes containing state of parallel computation, it does not face the problems in the COP model that arise out of failures of its participant processes. Stateless executors and the ability to locally or remotely re-execute RIBs free the programmer from handling failures at the programming level. The ARC model makes programs load adaptable. The dynamically growing nodes of a task tree may be suitably clustered and assigned to available machines. Transparency to distribution and heterogeneity makes it possible to load the tasks dynamically on suitable machines. Moreover, ARC provides an abstraction called horse power factor, which gives a measure of the current processing power of a machine. Abstracting load and speed (i.e. heterogeneity) in this way makes it possible for a parallel program to execute on unevenly loaded heterogeneous machines.

5 THE TWO-TIERED ARC LANGUAGE CONSTRUCTS

We introduce in this section, the two-tiered ARC language constructs. The lower layer provides primitives to support system tasks such as creation of RIBs, fault tolerance, load sensing and source code migration. The upper layer consists of higher level language extensions that advocate ARC programming methodology.

5.1 The Evolution of RIB

Execution of code at remote nodes is a widely used programming paradigm in distributed systems. As classified in Section 3, they fall into two categories of remote procedure calls and remote execution. Table 1 summarizes the main differences between the RIB approach and these paradigms.

RPC is perhaps the most popular remote execution paradigm used to build distributed applications. Many specific RPC protocols have been developed in recent years to meet the application specific requirements. For example, the blocking RPC [31] has been extended to MultiRPC, in which, the client can make calls simultaneously to multiple servers for locating files [34]. In the conventional RPC paradigm, the remote procedure is compiled and registered with the server process. In other words, the server exports a set of procedures that can be invoked by the clients. In the case of ARC, a program identifies several RIBs that may be executed on remote nodes concurrently. Thus the RIBs are part of client programs and not the procedures that servers export. RIBs can be migrated and executed on other nodes.

Migration of RIBs is different from the heavy weight process migration employed by operating systems for load balancing [1], [36]. However, a programmer may not distinctly benefit from process migration mechanisms unless the program is broken into a collection of concurrent processes as in the COP model.

The Remote Evaluation (REV) has been developed for providing migration of executable code to servers [37]. The main motivation in developing REV is to reduce the client-server communication. Instead of moving large amounts of data from servers to client for processing, the executable code is migrated to the server site and the results are sent back to the client. A similar approach can also be found in computation migration [23]. The computation is migrated to the data site to exploit the locality of data references. Though our approach resembles the REV and the computation migration in migrating the code to the remote node, the motivation is parallel programming on anonymous nodes providing fault tolerance, heterogeneity and load adaptability rather than distributed application development with well-known servers and clients.

RIBs are the basic building blocks of an ARC program. The lower layer ARC interface consists of a set of constructs which are used to specify and manipulate various parameters associated with RIBs. The distributed ARC kernel provides the run-time support for the RIBs. An RIB can be submitted, executed, and manipulated with the help of the ARC kernel interface calls.

5.2 The Design of the Lower Layer ARC Interface

The ARC lower layer provides support for execution of RIBs, fault tolerance, and load adaptability. An ARC distributed kernel implements these functionalities. The implementation of the kernel is described in the next section while the ARC interface is described in detail in this section. In this subsection, a detailed overview of the ARC Primitives (ARCPs) is provided. An RIB execution follows the following steps:
TABLE 1
THE EVOLUTION OF RIB

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Motivation</th>
<th>Execution</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Migration</td>
<td>Load balancing</td>
<td>Run-time migration of processes</td>
<td>Heavy migration cost, unsuitable for heterogenous architectures</td>
</tr>
<tr>
<td>Computation migration</td>
<td>Enhancing locality of references</td>
<td>Migrate computation accessing remote data</td>
<td>Does not address parallelism</td>
</tr>
<tr>
<td>RPC</td>
<td>Transparent access to remote procedures</td>
<td>Blocks the client when server executes the call</td>
<td>Placement of functions on servers is predetermined, parallelism not supported</td>
</tr>
<tr>
<td>multi-RPC</td>
<td>Multiple remote calls at a time</td>
<td>Multiple RPC calls made to servers</td>
<td>Used in locating files from multiple servers, function placement is predetermined</td>
</tr>
<tr>
<td>REV</td>
<td>To reduce client-server communication</td>
<td>Function migrates from client to server at run-time</td>
<td>Parallelism is not addressed, focus is on distributed system services, program is given control over location of processing</td>
</tr>
<tr>
<td>RIB</td>
<td>To support ARC paradigm, to integrate fault tolerance, load adaptability, scalability, heterogeneity from parallelism point of view at language level</td>
<td>Procedures migrated at run-time, servers are anonymous</td>
<td>Supports asynchronous computing on workstation systems, nested RIBs possible, run-time as well as compile time support is required</td>
</tr>
</tbody>
</table>

1) obtain a lock on an anonymous computing unit
2) pack the arguments to the RIB
3) post the RIB on the lock obtained
4) obtain the results of the RIB execution

Locks are to be obtained on anonymous remote nodes as a confirmation for execution. Depending on the number of anonymous computing units in the systems, the kernel releases locks. Locks are associated with a *horse power factor* (HPF). The HPF encodes load and processing capacity of an anonymous node. After obtaining a lock, an ARC program may dynamically post a larger or a smaller RIB grain depending on the lock value. After securing a lock, the arguments to RIBs are packed together and the RIB is posted to the kernel. The results of an earlier posted RIB can be obtained at a later point of time. The ARCPs are discussed in three groups namely, *startup and closedown* primitives, *RIB support* primitives and the HPF and asynchronous initiation primitives for load adaptability.

5.2.1 Startup and Closedown Primitives
Before generating the ARC calls, a program needs to identify itself with the ARC kernel. This is done using the ARC-startup() primitive. A call to this primitive establishes necessary connections and performs the necessary initializations. The kernel keeps track of open connections. The ARC-closeDown() primitive unlinks the program from the ARC kernel, disconnects already established connections, and informs the kernel to update its state.

5.2.2 Primitive Support for RIBs
This section explains the RIB primitives for packaging arguments, posting RIBs and receiving remote results. Remote instruction blocks are extracted from an ARC program and kept separately in the local file system. They are made available in the form of compiled code as well as in the form of source code. Depending on the location of the remote machine and its heterogeneous or homogeneous nature, the source code or the executable code is migrated.

*Primitives for packaging.* The openDataPack() primitive returns a new data-pack handler, in which arbitrary number of arguments can be inserted using the insertDataPack() primitive. Data is inserted into the data-pack pointed by the data-pack handler. Any number of insert calls can be made on a particular data-pack handler till the handler is closed for further inserts. A contiguous packet is prepared for all the inserted arguments by calling the primitive closeDataPack(). Every RIB prepares its new data packet with these three primitives.

*Primitives for RIB posting.* The postRib() primitive is used to post an RIB to the ARC kernel which can allocate the RIB to an earlier obtained lock. It is a nonblocking call, and it returns immediately with a work identifier. The work identifier can be used at a later point of time to obtain its results and also to alter the parameters related to the corresponding RIB. Its arguments are the RIB identifier, the argument packet handler, and parameters to control re-execution. If the remote node fails, the retries parameter can suggest re-execution on a possibly available anonymous node if the timeout for the RIB is not elapsed. The retries parameter gives the maximum number of retries to be made within the given timeout for the RIB. A specific fault tolerant behavior can be obtained with the help of this retry-till-timeout mechanism. These two parameters can be changed dynamically as explained subsequently.

The primitive obtainResult() is used to obtain the results of an earlier RIB posted for remote execution. It returns a structure containing a status value along with other information such as the actual result pointer and execution time for the successful execution.

The status value WORK_IN_PROGRESS is returned if the timeout is reached but the result is not yet available. In this case, the timeout value may be incremented with the help of a call to setParam(). If the timeout is not reached, the call blocks. Subsequently, if the RIB is successfully executed, the call returns with a status value RESULT_OBTAINED along with the actual result. In case of a failure of the remote node, if the timeout value for the RIB is not crossed and retries are warranted, it tries to obtain a lock on an anonymous node to resend the RIB for re-execution. In case of a successful resend, the call returns with a value FRESH_TRY. If an anonymous node cannot be contacted due to its unavailability or due to the timeout value, a value REM_FAILURE is
returned. In this case, the program is advised to execute the code by inlining or to try again at a later time for a remote execution.

Primitives for parameter setting. The setParam() primitive is used to set the control parameters for a particular RIB posting. The parameters that can be controlled are the timeout value and the number of maximum retries that can be made within the timeout period in the case of failures. The time count starts when the work is posted. In the case of failures, the ARC kernel internally manages a fresh execution of the ARC call as explained above. The values of the parameters can be altered dynamically. A fresh call to setParam() resets the parameters to new values. Timeout can be set to an arbitrarily large value of INFINITY.

5.2.3 Horse Power Factor and Asynchronous Intimations
Load adaptability is a major concern for ARC. ARC provides basic primitives to provide load adaptability to programs without enforcing the actual load distribution scheme at the kernel implementation level in contrast to earlier schemes such as the multilevel scheduling in CHORES run-time system [17]. Horse power factor and asynchronous intimations are the two primitives that help programs to become adaptable to variations in loads and speeds of machines in the clusters.

The Horse Power Factor (HPF) primitive. A horse power factor corresponding to a machine at a given time represents its processing capability at that time. It is a dynamically varying number. Various machines in the network are normalized by a benchmark program to obtain a relative index of speeds. This relative index is statically obtained. The HPF for a machine is dynamically obtained by considering this relative index and the load on that machine. The motivation behind the HPF is that, if differently loaded heterogeneous machines generate the same HPF figure, a task construct, the ARC function call models a sequential function invocation which may be executed at an anonymous remote node. An ARC function call can be made blocking or nonblocking. A nonblocking ARC function call is provided with a synchronizer in order to support concurrency outside the ARC call.

5.3 The Upper Layer ARC Constructs
The primitives explained above are built over the ARC kernel. A high level ARC language paradigm may use these primitives in various ways to provide easy to use ARC language constructs. We have earlier proposed Parsets and OBS, the two ARC language paradigms for procedural and object-oriented programming, respectively. Though these paradigms use a specific implementation different from the one described above, they are good examples of high level ARC paradigms. They use argument locking and object locking for synchronization. RIBs are function calls in the case of Parsets and meta message invocations called subcontracts in the case of OBS.

Below we present another example of a high level ARC construct, the ARC function call. While Parsets and OBS capture parallelism within a single method invocation or a single subcontract, the ARC function call models a sequential function invocation which may be executed at an anonymous remote node. An ARC function call can be made blocking or nonblocking. A nonblocking ARC function call is provided with a synchronizer in order to support concurrency outside the ARC call.

5.3.1 Blocking and Nonblocking ARC Calls
In this scheme, the user can tag the functions as belonging to one of the blocking or the nonblocking call classes. Examples for these calls are provided in Fig. 6. While generating the lower level ARC interface calls from these specifications, appropriate default values are set for timeout and

<table>
<thead>
<tr>
<th>TABLE 2</th>
<th>PERFORMANCE OF THE HORSE POWER FACTOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task: 200 × 200 Matrix Multiplication</td>
<td></td>
</tr>
<tr>
<td>Machine</td>
<td>No. of Users on the Machine</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>Pelican (Sun 3/50)</td>
<td>2</td>
</tr>
<tr>
<td>Bronto (Sun Sparc)</td>
<td>4</td>
</tr>
<tr>
<td>Pterano (Sun Sparc)</td>
<td>9</td>
</tr>
<tr>
<td>Vanavil (IBM RS/6000)</td>
<td>8</td>
</tr>
</tbody>
</table>
retries parameters. The blocking adaptive quadrature call may be executed on a remote anonymous node and the call returns only after successful completion. Internally, the runtime environment should manage the re-execution or inlining in the worst case. Default values for timeout and retries can be chosen as INFINITY and ZERO. The blocking ARC function call provides functionality similar to RPC, with a difference of the client migrating the function to be executed on an anonymous node. Instead of running a function on a slow host, it can be migrated transparently to a fast node. The blocking ARC function call is not designed for providing parallelism, but is targeted at providing speedup to a conventional sequential function call.

The nonblocking version of the ARC function call needs a synchronizer. While the call returns a synchronizer, it may be executing on a remote anonymous node. At a later point of time, the result of the nonblocking call can be obtained by waiting on the corresponding synchronizer. It can be seen that the synchronization scheme in Fig. 6 differs in a minor way from the scheme given in Fig. 4. In the example shown in Fig. 4, the high level ARC construct makes only a particular function invocation as an ARC call using RIB variables. Whereas, in this example, all invocations of a given function are treated as ARC calls. The postRib() function call described above is an example of a lower level nonblocking ARC function call. In this way it is possible to appropriate higher level ARC language semantics to suit the higher level requirements. More detailed higher level ARC language constructs are discussed elsewhere [25], [26].

6 IMPLEMENTATION

The anonymous remote computing mechanism is provided as an extension to C language. It is implemented on a heterogeneous cluster of workstations consisting of IBM RS/6000s, Sun Sparcs and Sun 3s. There are four IBM RS/6000 running AIX, three Sparcs running Solaris 2.3, and 18 Sun-3s running SunOS 4.0.3. In this section, we describe the implementation of ARC in this environment.

Fig. 7 depicts the architecture of the ARC implementation. A distributed ARC kernel is spread over the workstations that participate in anonymous remote computing. The entire domain is partitioned into three logical clusters having a separate file system for each one. The ARC kernel consists of multiple local coordinators to coordinate local activities and one system coordinator to coordinate the global activity. Each machine that intends to participate in the ARC system runs a local coordinator. The kernel supports the lower layer ARC primitives. It is possible to write ARC programs by directly using these primitives. Upper layer ARC programs such as those discussed in earlier sections can be converted into lower layer ARC programs consisting of these primitive calls.

6.1 The System Coordinator

There is only a single system coordinator in a given domain of logically grouped clusters. The system coordinator’s functions are to manage locks, route RIBs, and maintain migration history. It only functions as a policeman controlling and routing the traffic of ARC calls. It does not function as a task heap. In the following discussion, we discuss the various functions of the system coordinator.

Lock management. Whenever a machine wants to improve its utilization, it registers with the system coordinator through its local coordinator. The system coordinator then creates a free lock for this machine. A machine may deposit any number of free locks with the system coordinator through its local coordinator in order to boost

Fig. 6. Blocking and nonblocking ARC function calls.
its utilization. A wait queue is build for local coordinators seeking external RIBs through these free locks. Whenever a lock request arrives, a statistics-daemon is contacted to access the current load information for a given machine. From the current load information and the normalized speed of the machine, the HPF is computed and returned in response to the lock request.

Routing RIBs. An RIB posting is routed to the corresponding free local coordinator for execution. An RIB posting consists of the argument data packet, the source code or compiled code, whichever is appropriate, and the make directives for various architectures. In case of heterogeneity, the RIB source code is requested. Along with the source, the relevant make directives are also supplied. In our implementation, all homogeneous machines have access to RIB binaries through file system. Whereas, across heterogeneous machines the source code is migrated and RIB binaries are prepared for the target machine. The results of the RIBs are sent back to the corresponding local coordinators.

Maintaining migration history. A history of recent migrations is maintained. In the case of dynamically growing tasks, a machine may receive an RIB task belonging to an earlier posted source code. If an RIB source is already posted to a machine in a heterogeneous cluster, the binaries available in the file system of that cluster can directly be accessed. Thus, if an RIB with the same source arrives later for any machine in that cluster, repetitive migration and recompilation are avoided by using the history of recent migrations across clusters.

6.2 The Local Coordinator
The local coordinator runs on a machine that participates in the ARC system either to improve its utilization or to share its work with an anonymous remote node. Any ARC communication to or from the local processes is achieved through the local coordinator. The tasks of the local coordinator are described below.

Improve local utilization. An on-line command can be executed that directs the local coordinator to obtain a particular amount of work from the ARC system. This generates an asynchronous intimation that travels to the remote programs which may possibly use them to execute newly generated RIBs. When an RIB arrives, the local coordinator creates a task executor process corresponding to the RIB. Communication between a task executor and the coordinator is implemented using Unix Domain sockets. The task executor process receives the RIB arguments and executes the RIB code. Results are forwarded to the local coordinator. In the case of a dynamically growing task, the task executor is retained and another set of arguments for the same RIB may arrive. A task executor may also generate new RIBs. These RIBs can be further submitted for remote execution.

Accept RIBs from local processes. RIBs are initially generated by user processes. Later the RIBs themselves may generate new RIBs. A new RIB may be generated upon the receipt of an asynchronous intimation. The local coordinator provides the intimation to a program using a signal. When the results of remote RIBs arrive, they are queued up in a result queue. When a synchronizer in the program requires a particular result, the result queue is searched.

Fig. 7. The distributed ARC kernel.
6.3 The RiBs

RiBs are extracted out from the program using an ARC compiler. Following are the major tasks of the ARC compiler. An RiB code is generated using a generic RiB process that accepts arbitrary arguments and posts a result. Dynamic RiBs do not die immediately and can further accept new set of arguments. A specialized RiB is prepared from the generic RiB by integrating the required code fragments into it. Associated with each RiB is a set of make directives that dictate the compilation process on different available architectures. Whenever an RiB is migrated to a heterogeneous architecture, the appropriate make directive is also migrated. Using the make directive, the remote local coordinator compiles the RiB.

6.4 Timeouts and Fault Tolerance

The code that maintains the timeouts is integrated with the code that generates RiBs. A timer signal ticks at regular intervals. The *timeout* parameter in the lower layer ARC primitives is specified in terms of this interval. The current clock value is maintained in a variable that is accessed by the *obtainResult()* primitive to provide the timeout semantics as explained in the earlier section.

A remote node failure is detected by the system coordinator when local coordinator fails. The system coordinator maintains a list of incomplete jobs allocated to various local coordinators. Whenever a failure of a coordinator is detected, the RiB failure intimation is sent to the source program. A local coordinator may detect the failure of an executing RiB process, which might possibly occur due to program bugs. Such failures of an RiB are also reported back to the source program of the RiB. Upon detection of a failure, if the values of the *timeout* and *retries* parameters permit a re-execution, the RiB may be resent to another suitable remote node.

6.5 Security and Portability

Security in ARC implementation is built over the security provided by operating systems over which ARC is implemented. The system coordinator accepts connections only from local coordinators on trusted clients. An addition of a new machine into the system has to be registered with the system coordinator before it can be used to run ARC programs. A remote local coordinator accepting RiBs arriving from anonymous nodes runs with specific user permissions especially created for the purpose of ARC computations.

Portability to ARC programs is provided by making various ARC libraries portable across systems that are registered with the ARC system coordinator. Typically, the libraries that support the implementation of ARC primitives are made portable. However, the portability of user’s C code has to be ensured by the user.

When an RiB migrates to a heterogeneous node for dynamic compilation, the directives for the compiling environment are also migrated along with the RiB source code. An example compiling environment directive is the *-lm* option for compilation of an RiB that references functions supported by a portable *math.h* package.

7 Performance

The ARC approach provides an alternative to the conventional COP approach of parallel programming on a network of workstations. ARC provides for heterogeneity, fault tolerance and load adaptability to parallel programs. In this section we provide various performance tests to highlight these aspects of ARC.

7.1 Adaptability on Loaded Heterogeneous Workstations

This test was conducted to highlight the capability of an ARC program to run effectively on a heterogeneous system in the presence of load. The network consisted of IBM RS/6000s, Sun Sparcs and Sun 3/50s and Sun 3/60 workstations. The test was conducted for a traveling salesman problem using a variant of simulated annealing algorithm described by Ram et al. [24]. This problem was chosen for the adaptability test due to its high computation to communication ratio and dynamic task generation capabilities. The program starts with an initial solution space and an initial temperature. Locks are obtained on available anonymous processors dividing the initial solution space among them. Each processor carries out a number of iterations over each grain that it receives. As the results are brought back, the temperature is gradually reduced. As the temperature reduces, the tree is expanded by a factor of 2 to generate tasks dynamically. The tree can have smaller depths while it explores more breadth-wise. Each time a node finishes its task, a fresh lock is deposited by the node in order to obtain a new set of tasks. The ARC program assigns a varying number of tasks on a lock based on its judgment of the corresponding HPF. In every hop, a processor may get a different task size based on its current HPF. Obtaining a lock involves an RPC call by the system coordinator. The system under test consisted of two different networks, an FDDI ring of 100 MB/sec capacity and an Ethernet of 10 MB/sec capacity. The lock time was of the order of 100 msec on load conditions.

The results of the test are summarized in Tables 3 and 4. The load on the machines consisted of programs such as editing and compiling, simulation experiments generating CPU load, world wide web browsers generating heavy network traffic, and background batch jobs. We introduce the *horse power utilization* (HPU) figure to capture the utilization of a loaded heterogeneous multiprogramming multiprocessor environment. All the timing figures are quoted as the real time taken. The HPU for p processors and k heterogeneous clusters can be computed as:

$$HPU = \frac{\text{Total remote computation time} + \text{Total remote compilation time}}{p \times \text{Total execution time taken by the host program}}$$

$$= \frac{\sum_{i=1}^{p} Tc_i + \sum_{j=1}^{k} Tcompile_j}{p \times T_h}$$

where

- $T_h$ = Total time taken by the host program,
- $Tc_i$ = Compilation time taken by ith processor
- $Tcompile_j$ = Compilation time taken by jth heterogeneous cluster
JOSHI AND RAM: ANONYMOUS REMOTE COMPUTING: A PARADIGM FOR PARALLEL PROGRAMMING 87

includes the total time spent in obtaining locks which provide the current horse power factors for the corresponding locks. Table 3 shows the load shared by 16 heterogeneous workstations. Number of hops taken is the number of times a workstation obtained anonymous computing units. Average HPF per hop is indicated as a measure of availability of a workstation. The total number of TSP tree nodes processed by a workstation are indicated in the next column.

When a new task arrives at a heterogeneous cluster for the first time, the source code is transmitted for compilation. The compilation times are also mentioned in the table. While a task was getting compiled on a node in a cluster, other nodes in that cluster waited for the completion of the compilation process. The last column lists the computation time taken by each workstation. It can be seen that, in spite of large variations in speeds and loads of the workstations, a good balance was achieved. A horse power utilization (HPU) figure is computed as a measure of effectiveness of using the loaded heterogeneous workstations by a program. HPU includes all runtime overheads of the system. A load imbalance figure is computed in terms of the time spent in processing by each workstation. It gives an indication of the load balance achieved by the program using the HPF. Load imbalance is computed as the percentage average deviation from mean processing time.

\[
\text{Imbalance} = \frac{\sum_{i=1}^{p} |T_{C_i} - T_{C_{\text{mean}}}|}{pT_{C_{\text{mean}}}}
\]

TABLE 3

THE HETEROGENEITY LOAD TEST, 16 PROCESSORS

<table>
<thead>
<tr>
<th>Machine</th>
<th>No. of Hops Taken</th>
<th>Average HPF per Hop</th>
<th>No. of Tree Nodes Processed</th>
<th>Compilation Time (sec)</th>
<th>Processing Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM RS/6000</td>
<td>33</td>
<td>9209</td>
<td>831</td>
<td>1.31</td>
<td>514.89</td>
</tr>
<tr>
<td>IBM RS/6000</td>
<td>18</td>
<td>5088</td>
<td>441</td>
<td>waiting time</td>
<td>592.65</td>
</tr>
<tr>
<td>IBM RS/6000</td>
<td>19</td>
<td>5040</td>
<td>441</td>
<td>waiting time</td>
<td>581.80</td>
</tr>
<tr>
<td>IBM RS/6000</td>
<td>18</td>
<td>4916</td>
<td>440</td>
<td>waiting time</td>
<td>584.21</td>
</tr>
<tr>
<td>Sun Sparc</td>
<td>16</td>
<td>1127</td>
<td>286</td>
<td>20.55</td>
<td>571.44</td>
</tr>
<tr>
<td>Sun Sparc</td>
<td>14</td>
<td>959</td>
<td>220</td>
<td>waiting time</td>
<td>563.60</td>
</tr>
<tr>
<td>Sun 3/60</td>
<td>40</td>
<td>54</td>
<td>70</td>
<td>nil</td>
<td>540.92</td>
</tr>
<tr>
<td>Sun 3/60</td>
<td>39</td>
<td>53</td>
<td>67</td>
<td>nil</td>
<td>546.70</td>
</tr>
<tr>
<td>Sun 3/50</td>
<td>35</td>
<td>36</td>
<td>45</td>
<td>nil</td>
<td>574.08</td>
</tr>
<tr>
<td>Sun 3/50</td>
<td>34</td>
<td>36</td>
<td>44</td>
<td>nil</td>
<td>556.98</td>
</tr>
<tr>
<td>Sun 3/50</td>
<td>36</td>
<td>36</td>
<td>47</td>
<td>nil</td>
<td>565.92</td>
</tr>
<tr>
<td>Sun 3/50</td>
<td>30</td>
<td>35</td>
<td>39</td>
<td>nil</td>
<td>566.92</td>
</tr>
<tr>
<td>Sun 3/50</td>
<td>29</td>
<td>32</td>
<td>37</td>
<td>nil</td>
<td>557.10</td>
</tr>
<tr>
<td>Sun 3/50</td>
<td>30</td>
<td>32</td>
<td>38</td>
<td>nil</td>
<td>562.56</td>
</tr>
<tr>
<td>Sun 3/50</td>
<td>29</td>
<td>31</td>
<td>35</td>
<td>nil</td>
<td>554.16</td>
</tr>
<tr>
<td>Sun 3/50</td>
<td>17</td>
<td>24</td>
<td>19</td>
<td>nil</td>
<td>580.66</td>
</tr>
</tbody>
</table>

Total no. of locks obtained (total no. of hops) 437

Time spent in obtaining locks 135.24

Total Processing Time 661.84

HP Utilization 85.34%

Task Imbalance 0.15%

TABLE 4

LOAD ADAPTABILITY TEST

<table>
<thead>
<tr>
<th>No. of Processors</th>
<th>Machine Architectures</th>
<th>Tree Size in No. of Nodes</th>
<th>No. of Iterations per Tree Node</th>
<th>Average HPF per Hop</th>
<th>Load Imbalance (%)</th>
<th>HPU (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1 IBM RS/6000, 2 Sun Sparcs, 1 Sun 3/60</td>
<td>465</td>
<td>1500</td>
<td>1208</td>
<td>1.13</td>
<td>85.85</td>
</tr>
<tr>
<td>8</td>
<td>4 IBM RS/6000s, 2 Sun Sparcs, 2 Sun 3/60s</td>
<td>1550</td>
<td>2000</td>
<td>5429</td>
<td>0.43</td>
<td>86.40</td>
</tr>
<tr>
<td>12</td>
<td>4 IBM RS/6000s, 2 Sun Sparcs, 2 Sun 3/60s, 4 Sun 3/50s</td>
<td>2480</td>
<td>2000</td>
<td>2216</td>
<td>0.31</td>
<td>84.65</td>
</tr>
<tr>
<td>16</td>
<td>4 IBM RS/6000s, 2 Sun Sparcs, 2 Sun 3/60s, 8 Sun 3/50s</td>
<td>3100</td>
<td>2000</td>
<td>1427</td>
<td>0.15</td>
<td>85.34</td>
</tr>
</tbody>
</table>
Table 4 shows the results of the test conducted with increasing task sizes on increasing number of nodes. As we increase the task size, the HPU on a loaded heterogeneous system can be maintained at the same high level with increasing workstation processing power, similar to the trends in the conventional efficiency figure on no load parallel systems. However, our experimentation on dynamic systems shows that it can be extremely difficult to obtain good HPUs on loaded heterogeneous workstations, irrespective of task sizes, if good load balancing is not obtained. The *horse power factor* figure can be effectively used in a load balancing scheme that is developed to implement a given application. Table 4 lists 4 tests conducted on 4 to 16 workstations at different times. From average HPF per hop figure as listed in column 5, it can be observed that load in the network varied from test to test. For all the tests, a good HPU was obtained with low imbalance figures. The number of iteration per TSP tree node are also listed in the table along with the total number of tree nodes processed as an indication of the overall task size.

It can be concluded from these tables that ARC enables a parallel program to run in a network of heterogeneous workstations along with other existing sequential programs running in the system. The ARC parallel program has no control over the sequential programs that are started independently on different machines either from respective consoles or through remote logins. Instead, the ARC parallel programs adapt themselves to varying loads and speeds.

### 7.2 The Fault Tolerance Test

The fault tolerance test was conducted by artificially injecting faults by terminating remote processes. The results are summed up in Table 5. The *timeout* figure is used for deciding on re-execution in the case of a failure. It can be noted that *timeout* does not specify an upper bound on the completion of the task. Whenever a specified timeout is exceeded by a remote execution, a new execution is not tried automatically in the case of a failure. Instead, the failure is reported. However, if the timeout is not exceeded and a failure is detected, a re-execution is tried automatically. An upper bound on the number of retries is specified by the *retries* parameter.

As an example, in Table 5, the third test specifies a timeout of 1,000 units and a maximum of two retries. Two artificial failures were injected within the specified timeout, and the ARC posting was successful in its third attempt. By altering the timeout and retries parameters, a desired fault tolerance semantics can be obtained.

### 7.3 The Parallelism No Load Test

This test was conducted using a double precision matrix multiplication program on a network of homogeneous workstations on no load conditions to compare the overheads of ARC with a widely used traditional COP paradigm, the PVM. The results were compared with the same program implemented using PVM on no load. Table 6 summarizes the speedup figures obtained for various task sizes. It can be observed that ARC has slightly more overheads than PVM due to an additional message indirection through the ARC kernel. These overheads are due to the fact that ARC provides fault tolerance, load adaptability and automatic handling of heterogeneity not supported by the conventional COP approaches such as PVM.

### 7.4 ARC vs. Sequential Program

This test was conducted to show the system overhead in the case of total absence of remote anonymous workstations. In such a case, the program may still execute on the host node dynamically keeping track of availability through asynchronous intimations as described in Section 5.2.3. Table 7 shows the results of the no load inlining test conducted on a Sun 3, with a matrix multiplication program on a network of homogeneous workstations on no load conditions to compare the overhead of ARC with a widely used traditional COP paradigm, the PVM. The results were compared with the same program implemented using PVM on no load. Table 6 summarizes the speedup figures obtained for various task sizes. It can be observed that ARC has slightly more overheads than PVM due to an additional message indirection through the ARC kernel. These overheads are due to the fact that ARC provides fault tolerance, load adaptability and automatic handling of heterogeneity not supported by the conventional COP approaches such as PVM.
ous intimations thus help in reducing the number of unsuccessful calls to the system coordinator. It can be observed from the table that an ARC program can continue to execute on the host node without incurring significant slowdowns when the problem sizes are sufficiently large.

### 7.5 ARC Calls vs. RPC

This test was performed to estimate the overhead of making ARC function calls as compared to remote procedure calls. The tests were carried out on homogeneous machines. A major component in this overhead comes from the dynamic loading of the RIB for execution by the local coordinator on the remote node. A standard matrix multiplication program with varying sizes was executed using both RPC and ARC function calls. ARC function calls used the same nodes of the network as used by the RPC. ARC showed 4 to 5 percent overhead in terms of time as compared to its equivalent RPC call in which the code to be executed was already available at the known server. This additional cost is very nominal as ARC provides the flexibility of choosing varying number of appropriate nodes for execution at runtime. It can be noted that this overhead is incurred only once in a particular program when it makes multiple ARC calls as in the TSP problem discussed earlier. The ARC kernel recognizes the availability of an already active RIB code waiting to receive a subtask eliminating restarting of the RIB again on the remote node.

### 8 Conclusions

The specific issues involved in parallel programming on workstation clusters have been examined in detail. The Anonymous Remote computing (ARC) paradigm has been proposed for parallel programming on loaded heterogeneous cluster of workstations. The ARC model is developed in a two-tiered ARC architecture. The lower level consists of a cluster of workstations. The ARC model is developed in a paradigm for parallel programming on loaded heterogeneous computer systems.

### Acknowledgments

We thank the anonymous reviewers for their valuable comments and suggestions, which caused improvements in this work. We also thank Parthasarathy, RagHAV, and Vasan for building an alternative implementation for ARC, and Vaidehi for providing useful comments on the revised draft.

### References


---

**TABLE 7**

<table>
<thead>
<tr>
<th>Task Size</th>
<th>Pure Sequential (sec)</th>
<th>ARC on Same Node by Inlining (sec)</th>
<th>Slowdown (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>25 x 25</td>
<td>2.00</td>
<td>2.06</td>
<td>3.00</td>
</tr>
<tr>
<td>50 x 50</td>
<td>16.08</td>
<td>16.28</td>
<td>1.25</td>
</tr>
<tr>
<td>75 x 75</td>
<td>55.88</td>
<td>55.18</td>
<td>0.55</td>
</tr>
<tr>
<td>100 x 100</td>
<td>129.44</td>
<td>130.4</td>
<td>0.75</td>
</tr>
</tbody>
</table>


Rushikesh K. Joshi graduated from the College of Engineering, Pune, and received his doctoral degree from the Department of Computer Science and Engineering, Indian Institute of Technology Madras, in 1998. He is currently a faculty member in the Department of Computer Science and Engineering at the Indian Institute of Technology Bombay. His areas of specialization are distributed systems, object-oriented systems, and software engineering. His research is currently focused on evolving abstractions for software systems at the architectural and language level. His recent contributions are the ARC paradigm, Parsets and OBS abstractions, and a first-class filter object model for object-oriented systems.

D. Janaki Ram obtained his PhD degree from the Indian Institute of Technology Delhi, in 1989. Dr. Ram is currently with the Department of Computer Science and Engineering, Indian Institute of Technology Madras, Chennai, India, where he heads and coordinates the research activities of the Distributed and Object Systems Laboratory. He has taught several courses on distributed systems, object-oriented software development, operating systems, programming languages, and artificial intelligence at graduate and undergraduate levels. He has also to his credit several publications in leading international journals and conferences. He was program chair for the Eighth International Conference on Management of Data (COMAD’97) and the first National Conference on Object-Oriented Technology (NCOOT’97). He has served as a program committee member on several international conferences. He is also a consulting engineer in the area of object-oriented software development, organizational reuse for several software industries. His research interests include software engineering, distributed and heterogeneous computing, distributed object databases, and object technology. Dr. Ram is a member of the IEEE and the IEEE Computer Society.