MALLBA: Middleware for Communications in a Geographically Distributed Optimization System

Alba E., Cotta C., Díaz M., Soler E., Troya J.M.

May 30, 2000

Departamento de Lenguajes y Ciencias de la Computación Universidad de Málaga
{eat, ccottap, mdr, esc, troya}@lcc.uma.es

Abstract

This paper contains a preliminar study on the most appropriate communication middleware to be used in the MALLBA project. Since our goal is to deliver a user friendly and geographically distributed optimization system we must analyze several issues on the best ways of building such a middleware. First, we review the potential advantages and drawbacks of some existing communication models and tools. Then, we discuss on the services required by our optimization system and give a possible service definition for the middleware. Finally, the integration of the proposed middleware and the optimization skeleton being devised is studied with the aim of providing an integral solution to distributed optimization systems.

1 Introduction

This paper discusses the architecture and functionalities of a communication system that we call "middleware" to be used as the basic means for parallelizing and controlling the kind of processes needed in the project MALLBA (TIC1999-0754-C03-03).

Briefly stated, the MALLBA project is intended to provide a geographically distributed optimization system that allows a novice (though technical) user to pose a problem to be solved in parallel by many clusters of heterogeneous computers. Such a system could be build in many forms, but in MALLBA the optimization engines are embedded in the so-called "skeletons". A skeleton is a generic tool allowing the user to define a concrete optimization algorithm by creating instances of a general optimization procedure. The user is guided by a graphical interface to fill up the "holes" representing the alternatives the optimization algorithm provides to the user.

Skeletons for heuristic, exact, and hybrid algorithms will be developed within MALLBA, allowing a user to specify a problem-dependent algorithm that fits

his needs. Since the problems to which these skeletons will be faced can be quite difficult, parallel skeletons can also be specified, as extensions of the sequential ones. The parallel skeletons can run both on a NOW (Network Of Workstations) or in a geographically distributed system composed by several clusters of computers having different hardware capabilities.

After this brief revision of the MALLBA goals one point is clear: the design of the communication facilities for such a system is a very important issue. The communication services should be defined to be abstract enough for use in the skeletons and, at the same time, specific enough to the NOWs' technologies in order to get efficiency. By using the middleware services the skeleton designer will be able to spawn, trace, modify, and shut down the whole optimization task. This is the basic communication layer to be extended in order to build a more sophisticated service permitting automatic process location and taking into account the actual state of the whole hardware system (workstations, LAN and WAN links).

In short, we first need to review the existing communication models and toolkits in order to find out whether and how they match our requirements. Section 2 contains a brief discussion on this matter. Afterward, in Section 3, we present a first draft on the services of the middleware system. In Section 4 the integration between the optimization skeletons and the middleware library is analyzed. Finally, some conclusions and further work is pointed out in Section 5.

2 Evaluation of Existing Communication Systems

The principle objection to parallel computers and applications is that they are difficult to program. There is a significant component of truth in this claim, particularly for large-scale parallel machines. However, most scientific calculations require parallel computers to yield useful results in affordable run times.

There are three reasons why parallel programming is more challenging than serial programming [19]. First, parallel programs must include the mechanics of exchanging data between processors or handling mutual exclusion regions. Second, in an efficient parallel program the work must be evenly divided among processors. Third, the data structures must be divided among processors to preserve data locality. The first reason adds complexity to the semantics and the syntax of a program, the second one is an algorithmic challenge with no serial counterpart, and the latter one is an extension of serial data locality and cache performance issues.

In this section we review some programming tools that allow parallel programming at different levels of flexibility and generality. We then discuss in a final subsection the advantages and drawbacks of the outlined systems from the point of view of our geographically distributed system. Obviously, there is a large number of such communication facilities, from which only a representa-

tive subset is included here. Now, let us proceed with an overview of popular communication models and tools.

2.1 Sockets

The BSD socket interface (see e.g. [3])is a widely available message-passing programming tool. A set of data structures and C functions allow the programmer to establish full-duplex connections between two computers with TCP for implementing general purpose distributed applications. Also, a connectionless service over UDP (and even over IP) is available for applications needing such a facility. Synchronous and asynchronous parallel programs can be developed with the socket API, with the added benefits of large applicability, high standardization, and complete control on the communication primitives.

In despite their advantages, programming with sockets has many drawbacks for applications involving a large number of computers with different operating systems and on different networks. First, programming with sockets is error-prone and requires understanding low level characteristics of the network. Also, it does not include any process management, fault tolerance, task migration, security options, and other attributed usually requested in modern parallel applications.

In short, it is a great tool but its applications on modern internet and distributed systems requires a considerable effort to meet a satisfactory degree of abstraction.

2.2 PVM

The Parallel Virtual Machine (PVM) [14]is a software system that permits the utilization of a heterogeneous network of parallel and serial computers as a unified general and flexible concurrent computational resource. The PVM system initially supported the message passing, shared-memory, and hybrid paradigms; thus, allowing applications to use the most appropriate computing model for the entire application of for individual sub-algorithms. Processing elements in PVM may be scalar machines, distributed and shared-memory multiprocessors, vector computers and special purpose graphic engines; thereby, permitting the use of the best-suited computing resource for each component of an application.

The PVM system is composed of a suite of user interface primitives supporting software that together enable concurrent computing on loosely coupled networks of processing elements. PVM may be implemented on heterogeneous architectures and networks. These computing elements are accessed by applications via a standard interface that supports common concurrent processing paradigms in the form of well-defined primitives that are embedded in procedural host languages. Application programs are composed of *components* that are sub-tasks at a moderately larger level of granularity. During execution, multiple instances of each component may be initiated.

The advantages of PVM are its wide acceptability, and its heterogeneous computing facilities, including fault tolerance issues, and interoperability [15].

Managing a dynamic collection of potentially heterogeneous computational resources as a single parallel computer is the real appealing treat of PVM. In despite a large number of advantages, the standard for PVM has recently beginning to be unsupported (no further releases); also, users of the messaging-pass paradigm are shifting from using PVM to more new models of such paradigm that run more efficiently on the new kinds of networks appearing nowadays. In addition, since PVM 3.4 (the latest version) does not support threads, applications targeted to shared-memory computers do not use this software.

2.3 MPI

The Message Passing Interface (MPI) is a library of message-passing routines [13]. When MPI is used, the processes in a distributed program are written in a sequential language such as C or Fortran; they communicate and synchronize by calling functions in the MPI library.

The MPI application programmer's interface (API) was defined in the mid-1990s by a large group of people from academia, government, and industry. The interface reflects people's experiences with earlier message-passing libraries, such as PVM. The goal of the group was to develop a single library that could be implemented efficiently on the variety of multiple processor machines. MPI has now become the *de facto* standard, and several implementations exist, such as MPICH `www.mcs.anl.gov/mpi/mpich` and LAM/MPI `www.mpi.nd.edu/lam`.

MPI programs have what is called an SPMD style - single program, multiple data. In particular, every processor executes a copy of the same program. Each instance of the program can determine its own identity and hence take different actions. The instances interact by calling MPI library functions. The MPI functions support process-to-process communication, group communication, setting up and managing communication groups, and interacting with the environment.

The MPI standard allows programmers to write message-passing programs without concern for low-level details such as machine type, network structure, low-level protocols, etc. MPICH provides a portable, high-performance implementation of MPI that incorporates some support for heterogeneous environments (e.g. the `p4` device), but provides only limited support for wide-area metacomputing environments [7]. Some extensions by using the Nexus communication library are available to interface with the Globus metacomputing toolkit, an ongoing research project with important implications for the parallel computing community (see the next section).

The impetus for developing MPI was that each massively parallel processor (MPP) vendor was creating its own proprietary message passing API. In this scenario it was not possible to write a portable parallel application. MPI is intended to be a standard for message passing specifications that each MPP vendor would implement on its system. The MPP vendors need to be able to deliver high-performance and this became the focus of the MPI design. Given this design focus, MPI is expected to always be faster than PVM on MPP hosts [15].

The first standard named MPI-1 contains the following features:

- A large set of point-to-point communication routines, by far the richest set of any library to date.
- A large set of collective communication routines for communication among groups of processes.
- A communication context that provides support for the design of safe parallel software libraries.
- The ability to specify communication topologies.
- The ability to create derived data types that describe messages of non-contiguous data.

MPI-1 users soon discovered that their applications were not portable across a network of workstations because there was no standard method to start MPI tasks on separate hosts. Different MPI implementations used different methods. In 1995 the MPI committee began meeting to design the MPI-2 specification to correct this problem and to add additional communication functions to MPI including:

- `MPI_SPAWN` functions to start MPI processes.
- One-sided communication functions such as `put` and `get`.
- `MPI_IO`.
- Language bindings for C++.

The MPI-2 specification was finished in June 1997. The MPI-2 document adds 200 functions to the 128 original functions specified in the MPI-1.

All the mentioned advantages have made MPI the standard for the future applications using message-passing services. The drawbacks relating dynamic process creation and interoperability are being successfully solved. In addition, the connectivity with the Globus system is an important feature that stresses the importance of MPI.

2.4 GLOBUS

Globus is a new, extremely ambitious project to construct a comprehensive set of tools for building meta-computing applications [6]. The goal of the Globus project is to provide a basic set of tools that can be used to construct portable, high-performance services, which in turn support metacomputing applications. Globus thus builds upon and vastly extends the services provided by earlier systems such as PVM, MPI, Condor, and Legion. The project is also concerned with developing ways to allow high-level services to observe and guide the operation on the low-level mechanisms.

The toolkit modules execute on top of a meta-computer infrastructure and they are used to implement high-level services. The metacomputer infrastructure, or testbed, is realized by software that connects computers together. Two

instances of such an infrastructure have been built by the Globus group. The first, the I-WAY networking experiment, was built in 1996; it connected 17 sites in North America and was used by 60 research groups to develop applications. The second metacomputer infrastructure, GUSTO (Globus Ubiquitous Supercomputing Testbed), was built in 1997 as a prototype for a computational grid consisting of about 15 sites. GUSTO in fact won a major award for advancing high performance distributed computing.

The Globus toolkit consists of several modules:

- *Communication module*: Provides efficient implementation of many of the communication mechanisms, including message passing, multicast, remote procedure call, and distributed shared memory. It is based on the Nexus communication library.
- *Resource location and allocation module*: Provides mechanisms that allow applications to specify their resource requirements, locate resources that meet those requirements, and acquire access to them.
- *Resource information module*: Provides a directory service that enables applications to obtain real-time information about the status and structure of the underlying metacomputer.
- *Authentication module*: Provides mechanisms that are used to validate the identity of users and resources. These mechanisms are in turn used as building blocks for services such as authorization.
- *Process creation module*: Initiates new computations, integrates them with ongoing computations, and manages termination.
- *Access module*: Provides high-speed remote access to persistent storage, databases, and parallel file systems. the module uses the mechanisms of the Nexus communication library.

The Globus toolkit modules are being used to help implement high-level application services. One such service is what is called an Adaptive Wide Area Resource Environment (AWARE). It will contain an integrated set of services, including "metacomputing enabled" interfaces to an implementation of the MPI library, various programming languages, and tools for constructing virtual environments (CAVEs). The high-level services will also include those developed by others, including the Legion metacomputing system, and implementations of CORBA, the Common Object Request Broker Architecture. See [8] for more details on Globus, and the Globus Web site at `www.globus.org` for detailed information and current status of the project.

2.5 Java-RMI

The implementation of remote procedure calls (RPC) in Java is called Java-RMI [12]. The Remote Method Invocation in Java allows an application to

use a remote service with the added advantages of being platform-independent and able to access to the rest of useful Java characteristics when dealing with distributed computing and the Internet in general.

The client/server model used by JavaRMI is however somewhat slow in the current implementations of Java, an especially important consideration when dealing with optimization algorithms. An additional drawback is that current trends in the communication markets seem leading to abandon the support for this model of computation.

2.6 CORBA, ActiveX, and DCOM

Although many topics currently deal with multithreaded, parallel, and/or distributed programs, some higherlevel research is devoted on how to glue together existing or future applications so they can work together in a distributed, Web-based environment. Software systems that provide this glue have come to create the term "middleware". CORBA, ActiveX, and DCOM are three of the best known examples [17]. They and most other middleware systems are based on objectoriented technologies. Common Object Request Broker Architecture (CORBA) is a collection of specifications and tools to solve problems of interoperatibility in distributed systems (`www.omg.org`). ActiveX is a technology for combining Web applications such as browsers and Java applets with desktop services such as document processors and spreadsheets. The Distributed Component Object Model (DCOM) serves as a basis for remote communications, for example, between ActiveX components (`www.activex.org`).

CORBA is especially important because it is growing in the application side rapidly; it allows clients to invoke operations on distributed objects without concern for object location, programming language, operating system, communication protocols, or hardware. Reusability, interoperatibility, and the rest of mentioned advantages make CORBA a serious standard when dealing with objects in present distributed systems.

2.7 Others

There is an enormous number of other communication toolkits for constructing a new middleware system. Here, we only briefly review some of the most popular ones:

- **OpenMP**: OpenMP is a set of compiler directives and library routines that are used to express shared-memory parallelism (`www.openmp.org`). The OpenMP Application Program Interface (API) was developed by a group representing the major vendors of high-performance computing hardware and software. Fortran and C++ interfaces have been designed, with some efforts to standardize them. The majority of the OpenMP interface is a set of compiler directives. The programmer adds these to a sequential program to tell the compiler what parts of the program to execute concurrently, and to specify synchronization points. The directives

can be added incrementally, so OpenMP provides a path for parallelizing existing software. This contrasts with the Pthreads and MPI approaches, which are library routines that are linked with and called from a sequential program, and which require the programmer to manually divide up the computational work.

- **BSP**: The Bulk synchronous Parallel (BSP) model is a so-called bridging model that separates synchronization from communication and that incorporates the effects of a memory hierarchy and of message passing [18]. The BSP model has three components: processors, a communication network, and a mechanism for synchronizing all the processors at regular intervals. The parameters of the model are the number of processors, their speed, the cost of a communication, and the synchronization period. A BSP computation consists of a sequence of supersteps. In each superstep, every processor executes a computation that accesses its local memory and sends messages to other processors. The messages are requests to get a copy of (read) or to update (write) remote data. At the end of a superstep, the processors perform a barrier synchronization and then honor the requests they received during the superstep. The processors then proceed to the next superstep. In addition to being an interesting abstract model, BSP is now also a programming model supported by the Oxford Parallel Applications Center. More on BSP can be learned at `www.bsp-worldwide.org`.

- **Extensions to MPI**: While many hardware vendors have adopted the MPI standard and provide their own users with fast and stable implementations, there is no support for metacomputing with MPI for the moment being. PVM is designed to overcome that problem, but, since PVM is no longer the standard in the field most users have moved to MPI. To avoid changing the code of these users for metacomputing experiments PVMPI [5] has come to bridge the gap between PVM and MPI. But, in practice, this would require the user to substantially change his code. Only an interoperable MPI such as PACX-MPI [4] has finally provided access for metacomputing with MPI. A very promising wide-area implementation of MPI using the services in Nexus to interface Globus is the result of ongoing works [7].

- **Legion**: The Legion [9] research project at the University of Virginia (`www.virginia.edu`) aims to provide and architecture for designing and building system services that present the illusion of a single virtual machine. Persistence, security, improved response time, and greater throughput are among its many design goals. But, the key characteristic of the system is its ambition of presenting a transparent, single virtual machine interface to the user. Legion aims at presenting a seamless computing environment with a single namespace, but supporting multiple programming languages (and models) and interoperability. It is an objectoriented system that attempts to exploit inheritance, reuse, and encapsulation; the

distributed object programming system Mentat (a precursor to Legion) is in fact the basis for programming the first public release of Legion.

- **NetSolve**: A somewhat different, clientserverbased approach is adopted by NetSolve [2], a computational framework that allows users to access computational resources distributed across the network. NetSolve offers the ability to search for computational resources on a network, choose the best available resource based on a number of parameters, solve a problem (with retry for fault tolerance) and return the answer to the user. Resources used by NetSolve are computational servers that run on different hosts, and may provide both generic and specialized capabilities. The system provides a framework to allow these servers to be interfaced with virtually any numerical software. Access is achieved through a variety of interfaces; two which have been developed are as a MATLAB interface and a graphical Java interface. It is also possible to call NetSolve from C or FORTRAN programs using a NetSolve library API.

An interesting revision of heterogeneous distributed programming tools (including Globus, NetSolve, Harness, Legion, PVM, MPI, etc) can be found in (Sunderam and Geist 99).

2.8 Which of Them?

Since our goal is to construct an optimization system that spawns across several clusters of machines we can identify the following important issues that must be met by the middleware system:

- To have some abstract mechanisms to allow highorder communication operations by means of a rich programming interface.
- To be popular enough and extensively tested for becoming a good choice, not only at present but for future trends in communication models and in networking technology.
- To be general enough for highlevel programming but not too specialized in some kinds of applications, since our optimization problems can belong to quite different problem classes.
- To be targeted to a wide spectrum of applications and, at the same time, being able to provide efficiency in concrete clusters of machines.

Several other minor issues can be added to the previous list, but in practice the presented ones help in clarifying the decision. Since no one of the studied systems is by itself useful for our skeleton approach we need to select a communication toolkit for the implementation of our own middleware system. The reason is that interfacing the middleware with the optimization skeletons is a specialized operation that is not present in any of the mentioned tools.

Although we have made a somewhat detailed presentation of many systems, only a few of them fit our requirements. First, the socket abstraction for message

passing is powerful enough for our application domain, but it needs excessive low level manipulations when programming distributed systems with them. We discard other tools like JavaRMI since such a client/server model is still quite slow in present releases of Java; in addition, it seems that in the forthcoming years this standard is going to be abandoned in favor of other new models.

On the other side, CORBA seems the most widely accepted object broker for distributed systems. CORBA is a highly standard and crossplatform facility with the added advantage of being languageindependent. However, it is still unclear if our skeleton engine will present a truly objectoriented interface to its environment. This last, and the expected reduction in efficiency if the standard CORBA is chosen, both in a single cluster of machines and in a wide area network, are the reasons for not using it in the present implementation of the middleware for our applications.

We have reviewed also many other distributed programming paradigms such as BSP, OpenMP, and many tools for metacomputing such as Legion and others. None of them will be used for our middleware system since their close relationship with well established types of applications (shared memory systems, virtual computers with easy use but with a difficult control to achieve efficiency, ...). NetSolve seems appropriate at first glance. It allows the user application to call remote procedures such as FFT, matrix operations, etc. with some blocking/non blocking options. However, this can be useful in some engineering computations only, because it does not allow to control processes nor hardware capabilities.

Then, our discussion is reduced to three communication tools, namely PVM, MPI, and Globus. We first discard Globus since it has an excessively ambitious spectrum of applications. Globus is quite complex for our needs: we need controlling processes and exchanging messages among workstations located in the same cluster or among geographically distributed clusters. However, we are not still able at present of evaluating the importance of Globus in parallel computing, since its popularity, efficiency, and controllability are continuously growing in these days. Nonetheless, it would be great that the selected underlying communications could be carried out with a toolkit having present or scheduled future interface with Globus.

Finally, we end with two systems, PVM and MPI. Although and heterogeneous computing system could be readily implemented on PVM, we choose MPI for several reasons. First, the message passing community has shifted from PVM to MPI naturally, due to the largest efficiency that can be got with MPI. Second, PVM is beginning to enter the "unsupported status", this being a serious drawback for future users and projects based in PVM. Third, the initial problems in MPI relating interoperability, dynamic process groups and some other minor details are being solved in MPI-2 and the new standards. Fourth, MPI has recently being used in many works to be explicitly extended for efficient implementations in wide area networks. Fifth, MPI is quite popular, available, and standardized. Finally, MPI has already a direct extension to meet the Globus project through the Nexus services.

All these reasons lead us to rely on MPI for constructing the specialized middleware needed to be interfaced with the optimization skeletons for general

and, at the same time, efficient distributed optimization. The next section will draw the main services needed for the middleware system and then a forthcoming section will deal with how it is planned to use this middleware from inside the skeleton implementations.

3 An Initial Draft for the Middleware

Here follows a brief description of the communication interface to be used by the MALLBA project. Primitives can be grouped in several sets:

1. Process Execution.
2. Connection Management.
3. Miscellaneous.

Let us now proceed to discuss these primitives.

PROCESS EXECUTION

```
pid create_process(host, type, continue)
```

Create a process on a host. If the kind of process is known beforehand then the creation can be internally made more efficiently; otherwise, i.e. if there is a new kind of process or nothing is known about it a standard spawning is performed. The created process can be completely new or else to resume the work just abandoned by other process of the same type. This last allow to simulate the migration of a process to the user of the middleware. It is assumed that the remote host has the binary and configuration files to run such a process type. Upon successful completion this service returns the identifier of the created process (negative if an error occurs). Process ID's are unique in the system.

`host` is the IP address, DNS name or URL pointer to the host receiving this process. Using different names will allow to deal with the same physical host having different software capabilities. In addition, by generically using the Unified Resource Location (URL) we use an Internet standard to specify the domain, the host, and the protocol to access it.

`type` is a value from the set `(EA, DP, SA, DC, TS, OTHER)`. Having information from the kind of algorithm can enhance its parallel creation. If no information is available on the kind of algorithm the `OTHER` label can be used.

`continue` contains the process ID from which to continue execution (process migration occurs). If `null` then a new process will be created.

```
pstatus get_process_status(pid)
```

Gets the status of a running process. This status is a set of values for the next attributes (`status`, `step`, `best_solution_found`, `running_time`, `wasted_effort`).

The status of a process can be `creating`, `running`, `migrating`, `terminated`, or `unknown`.

`pid` is the connection associated to the remote process to get info from.

```
pinfo get_process_info(pid)
```

Gets the available information on a process. This information includes the `type` of process, its `pstatus`, and its `URL`.

`pid` is the remote process whose info is being requested.

`pinfo` is made up of type, status and URL.

```
int terminate_process(pid)
```

Kill a process in a controlled fashion. The `pid` is no longer valid in the process name space. Returns 0 upon successful termination, and a negative value if an error occurs, e.g. if the `pid` does not exist.

CONNECTION MANAGEMENT

```
cid open_connection(pid, copt)
```

This primitive creates a connection to a running process with a given connection model and with some optional flags. The process connected to must exist. It returns a connection ID (negative if an error occurs).

`pid` is the identifier of the process to connect to.

`copt` is a list with options used for the initial configuration of the channel. The following facilities will be available in the option list: `sync/async read/write` operations, `connection_oriented/connectionless`, and `data/control` connections.

```
int close_connection(cid)
```

Closes an open connection. It fails (negative integer) if the connection was not open. Future data operations on the closed connection will fail; `cid` is the connection to be closed.

```
long int send(cid, data, length)
```

Sends user data on an open connection by using the options defined when the connection was open. It returns the number of bytes sent on the connection (negative if an error occurs).

`cid` is the connection to transport data.

`data` is the information to be delivered.

`length` is the number of data bytes to be sent.

```
long int receive(cid, data, length)
```

Gets info from the communication channel. It returns the number of bytes read from the connection (negative if an error occurs).

`cid` is the connection to read the data from.

`data` is used to store the read data;

`length` is the number of bytes read from the connection.

```
int probe(cid)
```

Returns whether there is (1) or not (0) any info awaiting in the connection `cid`. If the returned value is negative an error has occurred, indicating e.g. unknown connection.

```
int connection_is_open(cid)
```

Gets the status of the connection: TRUE(1) if open and FALSE(0) if closed. A negative value indicates that the connection is unknown.

```
cstats get_connection_stats(cid)
```

Gets statistics on the connection.

`cid` is the connection to get info from.

verb"cstats" is a list of values including: number of bytes sent/received, round trip average time, etc.

```
model get_connection_model(cid)
```

Gets the model defining the actual behaviour of a connection. A connection model is a set of representative values for the connection between two processors. The model is computed by taking into account the model for the link between the two computers hosting the applications and the number and type of connections sharing the same physical link.

`model` is a list of values defining some connection features. Basically it contains a weekly mean (expected) model on the performance of the cluster and (WAN) links. The model can receive a `null` value, thus indicating that nothing is known about the connection. The information, when it exists, is structured by week day (from `monday` to `sunday`) and by hour (from 0 to 23). The specified information will contain the link `bandwidth` and `latency` at this moment.

MISCELLANEOUS

```
model get_link_model(url1, url2)
```

Gets the model defining the mean expected behaviour of a link between two URLs. This information is basically static for every link. The administrator is the responsible for feeding more exact values from time to time. This is the basic model that the middleware will use to compute the actual connection models.

`model` is a list of values defining `bandwidth` and `latency` weekly or `null` if no model is available.

`url1, url2` are URLs to two computers known to the system.

`url_list preferred_processors(type)`

This is a service computing an ordered list of preferred processors for a new process to be created. This list is worked out by considering the *computational load* of every processor and the *distance* to the actual processor running this middleware call. The processor load will be computed after its percentage of `CPU` and `memory` use, and other attributes internally considered important in future releases of the middleware. The distance between two processors is measured after the model of the link connecting this two processors (further tuning will be available in subsequent releases of the middleware).

Upon successful completion the result is a list of valid URLs for the middleware. If any error exists, or if processors cannot be ranked then a `null` value is returned.

4 Integration with the Optimization Skeletons

In this section we discuss an initial proposal for making the middleware system available for use in the optimization skeleton implementations. The middleware system is composed of several elements:

- The Application Program Interface (API).
- The System Files.
- The Instantiation Module (IM).

The API allows the skeletons to use the middleware. The system files contains information about the interconnected LANs, the processors, and the WAN as a whole. The instantiation module provides a way for the administrator to tune the behavior of the middleware for its own cluster.

We now proceed to explain their use in the following subsections.

4.1 The Application Program Interface (API)

The API for the skeletons to use the distributed system is defined by the set of data types and methods described in the previous section. By means of these data types and methods any skeleton will be able to spawn, change, terminate, and in general manage a complex system of parallel optimization skeletons.

All the skeletons, exact, heuristic, and hybrid must use the same API to access the network facilities. This will ensure having a similarly constructed set of optimization skeletons. The special needs of every kind of algorithm will be taken into account by accessing the flags and other indicators included to customize the well-known behavior of every primitive in the API.

Sequential skeletons will not use the API, unless a distributed optimization skeleton is requested to run on a single processor. Therefore, the middleware will be of interest for the development of distributed optimization skeletons.

This basic API includes simple services to be combined in the future into more complex services. This layered approach is quite usual in communication protocols (e.g. OSI, TCP/IP, etc.). Some more complex services such as automatic mapping of processes to processors, load balancing, and dynamic changes in the network of communicating skeletons will be addressed by combining the API primitives, and may be by extending their semantics and by adding new services in future versions of the middleware.

4.2 The System Files

Since our distributed optimization system is intended to run on a cluster of machines in a second phase of the MALLBA project (after the initial core-sequential phase) we need a model of the local area network. The reason is that we do not only want a distributed skeleton system, but an *efficient* distributed skeleton system. This efficiency requirement clearly makes necessary to take into account the special characteristics of the LAN being used while preserving the abstraction and standardization of the API.

Our proposal consists in providing LAN and WAN information into some systems files. The instantiation module will help the user to refine some general values to obtain an improved model of the communication network.

Internal models for most common networks (e.g. fast-ethernet, ATM, Mirynet) will be included for users having no idea on technical issues concerning their own network. At least, when installing the system, the kind of network to be used will be detected to get a good efficiency in most cases .

Since we are going to deal with very different LANs the LAN system file will contains the following data:

- **Name of the LAN**. Some default names will be provided, such as `ethernet, fast-ethernet, gigabit-ethernet, atm, mirynet, fddi, other`.
- **Theoretical bandwidth**. The default names will have internally an associate bandwidth depending on the official standards of the included networks, such as 155 Mbps for `atm` and 100 Mbps for `fast-ethernet`.
- **Number of processors**. The total number of processors to be used must be indicated beforehand. The processors to be used will be taken from a list of available processors by following the order in which this list is provided. Later in this section we explain how the processors will be managed.

The available set of processors needs to be defined by the user in order the system to know where to run the necessary optimization skeletons. Processors will be organized in clusters. Every cluster will contain thus a variable number of processors. For each processor the following information is needed:

- The DNS name of the processor.
- The IP address of the processor.

- The clock speed, if known.
- The RAM memory, if known.
- The operating system version, if known.
- Whether the processor is dedicated or not, if known.

At least, the name and/or the IP address must be given for every processor in order to be usable. The rest of parameters will have default values that can be tuned depending on the LAN knowledge the user has. The result of the LAN skeleton is a two-level hierarchy of machines and some additional info from which the API will take advantage to run more efficiently. The LAN file will validate at last some of the most critical values provided by the users, such as the name/address of every processor, in order to assess a minimum quality for the running environment.

As well as the LAN system file will help in a more efficient instantiation of the parallel optimization skeletons, a system file will exist to specify a model for the whole distributed system. After the definition of the LAN, we can easily see that the distributed system is considered as a set of machine clusters. At the geographically distributed level we specify some values concerning the whole set of computational resources to run a given skeleton.

The machines involved in the computations belong each one to one (and only one) given cluster. Thus, we need to know which clusters are considered and how they are linked. The considered parameters effecting the whole communication system are:

- A set of pairs indicating two interconnected clusters.
- For every pair of interconnected clusters a performance model will be given.
-

The performance model can be a `null` model, a `default` model, or a `user-defined` model. In any case the performance model will detail the following *week* information:

- A valid label indicating the day of the week (from `monday` to `sunday`).
- A beginning and terminating day hour (0..24h).
- The expected bandwidth and latency available at this day and time range.

4.3 The Instantiation Module

The Instantiation Module (IM) is a tool providing the user with the ability to get into the middleware system as much knowledge as possible about the processors, clusters, and WAN facilities.

The IM is the interface application that the user can invoke to instantiate and customize the automatic default behavior of the middleware system. It is expected that the performance of the optimization skeletons run in parallel will raise as the tuning of the middleware is higher to deal with the existing hardware.

5 Concluding Remarks

In this first draft we have discussed the pro and cons of many communication toolkits from the point of view of the MALLBA necessities. Our conclusion is that MPI is the standard that better fits our needs. These needs can be briefly summarized in the following topics: availability, efficiency, and future trends. Some other candidates such as CORBA or Globus have been finally left out at this stage of the MALLBA project.

With the LAN and WAN system files, the proposed API will provide a useful set of services to program distributed optimization algorithms, whatever the kind of skeleton is (exact, heuristic, or hybrid). More in dept study of this proposal will probably reveals that further tuning is needed, and this is the reason of presenting an architectural approach to the middleware system.

References

[1] Andrews G.R.(2000) "Foundations of Multithreaded, Parallel, and Distributed Programming. *Addison-Wesley*.

[2] Casanova H, Dongarra J.J. (1995) "NetSolve: A Network Server for Solving Computational Science Problems". *TR CS-95-313*, Univ. of Tennessee (November).

[3] Comer D.E., Stevens D.L. (1993) "Internetworking with TCP/IP (Volume III)". *Prentice-Hall*.

[4] Eickermann Th., Henrichs J., Resch M., Stoy R., Völpel R. (1998) "Metacomputing in Gigabit Environments: Networks, Tools, and Applications". *Parallel Computing* 24:1847–1872.

[5] Fagg G.E., Dongarra J.J. (1996) "PVMPI: An Integration of the PVM and MPY Systems". Department of Computer Science, *TR CS-96-328*, Univ. of Tennessee.

[6] Foster I., Kesselmann C.(1997) "Globus: A Metacomputing Infrastructure Toolkit". *Int. Journal of Supercomputing Applications* 11(2):115–128.

[7] Foster I., Geisler J., Gropp W., Karonis N., Lusk E., Thiruvathukal G., Tuecke S. (1998) "Wide-Area Implementation of the Message Passing Interface". *Parallel Computing* 24:1735–1749.

[8] Foster I., Kesselmann C. (1999) "The Grid: Blueprint for a New Computing Infrastructure". *Morgan Kaufmann*.

[9] Grimshaw A.S., Wulf W.A. (1997) "The Legion Vision of a Worldwide Virtual Computer". *Communications of the ACM* 40(1):39–45.

[10] Gropp W., Lusk E. (?) "Why are PVM and MPI so Different?" *TR* Mathematics and Computer Science Division, Argonne National Laboratory.

[11] Haupt T., Akarsu E., Fox G. (2000) "WebFlow: a Framework for Web Based Metacomputing". *Future Generation Computer Systems* 16:445–451.

[12] JavaSoft (1997) "RMI: The JDK 1.1 Specification". `javasoft.com/products/jdk/1.1/docs/guide/rmi/index.html`.

[13] Message Passing Interface Forum (1994) "MPI: A Message-Passing Interface Standard". *International Journal of Supercomputer Applications* 8(3/4):165–414.

[14] Sunderam V.S. (1990) "PVM: A Framework for Parallel Distributed Computing". *Journal of Concurrency Practice and Experience* 2(4):315–339.

[15] Sunderam V.S., Geist G.A. (1999) "Heterogeneous Parallel and Distributed Computing". *Parallel Computing* 25:1699–1721.

[16] Tierney B., Johnston W., Lee J., Thompson M. (2000) "A Data Intensive Distributed Computing Architecture for "Grid" Applications". *Future Generation Computer Systems* 16:473–481.

[17] Umar A. (1997) "Object-Oriented Client/Server Internet Environments". *Prentice-Hall*.

[18] Valiant L.G. (1990) "A Bridging Model for Parallel Computation". *Communications of the ACM* 33(8):103–11.

[19] Womble D.E., Sudip S.D., Hendrickson B., Heroux M.A., Plimpton S.J., Tomkins J.L., Greenberg D.S. (1999) "Massively Parallel Computing: A Sandia Perspective". *Parallel Computing* 25:1853–1876.