WAMM (Wide Area Metacomputer Manager): A Visual Interface for Managing Metacomputers
Version 1.0

Ranieri Baraglia, Gianluca Faieta, Marcello Formica, Domenico Laforenza

CNUCE - Institute of the Italian National Research Council
Via S.Maria, 36 - I56100 Pisa, Italy
Tel. +39-50-593111 - Fax +39-50-904052
email: R.Baraglia@cnuce.cnr.it, D.Laforenza@cnuce.cnr.it
meta@calpar.cnuce.cnr.it

Contents

1 Introduction

Last years have seen a considerable increase in computer performance, mainly as a result of faster hardware and more sophisticated software. Nevertheless, there are still problems, in the fields of science and engineering, that are not affordable using currently available supercomputers. Infact, these problems, due to their size and complexity, require a computing power which is considerably higher than currently available in a single machine. To deal with such problems, several supercomputers need to be concentrated on the same site to get the total power needed. This is obviously unfeasible both in terms of logistics and economics.

For a few years in some important research centers (mostly in the USA) tests have been made on the cooperative use, via network, of geographically distributed computational resources. Several words have been coined in connection with this approach, such as Metacomputing [1], Heterogenous Computing [3], Distributed Heterogeneous Supercomputing [2], Network Computing, etc. One of the main reasons for introducing computer networks was to allow researchers to carry out their work where they pleased, by giving them access to geographically distributed computing tools, both rapidly and transparently. Technological advances and the increasing diffusion of networks (originally used for file transfer, electronic mail and then remote login) now make it possible to achieve another interesting goal: to consider multiple resources distributed over a network as a single computer, that is, a *metacomputer*.

1.1 Metacomputer

A metacomputer is very different from a typical parallel MIMD-DM machine (e.g. Thinking Machine CM-5, nCUBE2, IBM SP2). Generally, a MIMD computer consists of *tightly coupled* processing nodes of the same type, size and power, whereas in a metacomputer the resources are *loosely coupled* and heterogeneous. Each of these can efficiently perform specific tasks (calculation, storage, rendering, etc.), and, in these terms, each machine can execute a suitable piece of an application. Thus, it is possible to exploit the *affinity* existing between software modules and architectural classes. For example, in a metacomputer containing a Connection Machine (CM-2) and an IBM SP2, an application which is partitionable in two components — one *data parallel*, the other *coarse-grain task farm* — would naturally exploit the features of both machines.

Metacomputing is now certainly feasible and could be an economically viable way to deal with some complex computational problems (not only technical and scientific ones), as a valid alternative to extremely costly traditional supercomputers. Metacomputing is still at an early stage and more research is necessary in several scientific and technological areas, for example:

1. methodologies and tools for the analysis, parallelization and distribution of an application on a metacomputer;
2. algorithms for process-processor allocation and load balancing in a heterogeneous environment;
3. user-friendly interfaces to manage and program metacomputers;
4. fault-tolerance and security of the metacomputing environments;
5. high performance networks.

2 Metacomputing environments

Developing metacomputing environments entails resolving several problems, both hardware (networks, mass memories with parallel access, etc.) and software (languages, development environments, resource management tools, etc). Although many of the hardware problems are close to a solution, software problems are still far from being resolved.

Currently available development environments generally have tools for managing the resources of a metacomputer, but often do not have adequate tools for designing and writing programs. Without such tools, software design cycle for metacomputers can come up against considerable difficulties. Typically, building an application for a metacomputer involves the following steps:

1. user writes the source files on a local node of the metacomputer;
2. source files are then transferred on every node;
3. the related compilation is made on all nodes;

4. if errors are detected or modifications made, then all the previous steps are repeated.

Compilation is needed on each node since it is not possible to determine a priori on which machines the modules that make up the application will be run. If the right tools are not available then the user has to manually transfer source files and execute the corresponding compilation on all the nodes; such operations have to be repeated whenever even the smallest error needs to be corrected. Therefore, even with just a few machines, a method to make these operations automatic is essential.

Some metacomputing environments provide tools to control certain aspects of configuration and management of the virtual machine, such as the activation, insertion and removal of nodes (e.g. the PVM console [11]). In certain cases, however, easier to use management tools are needed, above all when large metacomputers with several nodes are being worked on.

To alleviate these problems, we have developed a graphical interface based on OSF/Motif [21] and PVM, which simplifies operations normally carried out to build and use metacomputer applications, as well as for managing parallel virtual machines.

3 Design goals

This section describes the guidelines we followed in designing the interface. We believe they are general enough to be applicable to any development tool for metacomputing.

Easy of use of a metacomputer. The main aim of the interface should be the simplification of the use of a metacomputer. This entails giving the user an overall vision of the system, especially if there are many nodes spread out over several sites. At the same time, the individual resources should be easily identifiable. Although a simple list of the network addresses of the machines would probably be the fastest method to identify and access a particular node, it would be better to group machines by following some precise criteria so as to facilitate user *exploration* of the resources available on the network.

In addition, the interface should let users work above all on the local node. Operations that need to be carried out on remote nodes should be executed automatically. Thus, developing software for metacomputers will mainly require the use of the same tools used to write and set up sequential programs (editor, `make`, etc.). This way, the impact with a new programming environment will be less problematic.

Any simplifications of the use of metacomputers cannot be made if the tools themselves are not easy to use and intuitive. It is well known that Graphical User Interfaces (GUI) have gained the favor of computer users. We therefore decided to develop our interface as an X11 program, thus allowing users to access functionalities via windows, menus and icons. This requires the use of graphical terminals, but it saves users from having to learn new commands, keyboard shortcuts, etc.

System control. When working with a metacomputer, especially if a low level programming environment such as PVM is used, it may be difficult to control operations that occur on remote nodes: unexperienced users could be discouraged. An interface for programming and using a metacomputer should offer users as much information as possible, and full control on what happens in the system. For example, users should never get into situations where they do not know what is happening on a certain node. If problems arise, these should be communicated with complete messages and not with cryptic error codes. If the problem is so serious that the interface can no longer be used, then the program must exit tidily.

Virtual Machine management. The interface must have a set of basic functions to manage the virtual machine (addition/removal of nodes, control of the state of one or more nodes, creation and shutdown of the virtual machine). Essentially, all the basic functions of the PVM console should be implemented.

Process management. Again, the functionalities to implement should be, at least, the ones that are provided by the PVM console. It must be possible to spawn processes; the interface must allow the use of all the activation parameters and flags that can be used in PVM. Users should be able to redirect the output of the tasks towards the interface, so that they can control the behaviour of the programs in "real time".

Remote commands. When several machines are available, users often need to open sessions on remote hosts or, at least, execute remote commands (e.g.

`uptime` or `xload` so as to know the machine load). Using UNIX commands such as `rsh` to execute a program on a remote host is rather inconvenient, so the interface should simplify this. Further simplification is needed for X11 programs. When an X11 program is run on a remote host, windows have to be visualized on the local graphical terminal, which itself must be allowed to accept them. The `xhost` command is used to permit this, but it should be made automatic if X11 programs are run by the interface.

Remote compilation. One of the most important functionalities of the interface should be the ability to carry out the compilation of a program on remote machines. Once the local directory with the source codes has been specified, along with the `Makefile` to use and the hosts where the program has to be compiled, the remainder should be completely managed by the interface. This involves sending source files to remote hosts and starting compilers. Any such operations carried out by hand, apart from being time expensive, are also error prone.

Remote compilation is quite complex. The user of the interface must be able to follow the procedure step by step, and if necessary stop it at any moment. For users to feel at ease with an automatic tool, all the operations should be carried out tidily. For example, temporary old files, created on file systems of remote machines as a result of previous compilation, must be deleted transparently.

Configurability. The interface must be configurable so that it can be adapted to any number of machines and sites. The configuration of the metacomputer must therefore not be hard-coded in the program, but specified by external files.

To modify any graphical element on the interface (colours, window size, fonts, etc.) resource files should be used. This is the standard technique for all X11 programs, and does not require the interface to be re-compiled. Also, using the X11 program `editres`, graphical elements can be modified without having to write a resource file.

Finally, the program should not impose any constraints on the number of nodes and networks in the system, nor on what type or where they are.

Figure 1: WAMM

4 WAMM

On the basis of the goals and criteria defined above, we have developed WAMM (Wide Area Metacomputer Manager), an interface prototype for metacomputing (fig. 2). WAMM was written in C; in addition to PVM, the OSF/Motif and xpm[1] libraries are required. This section gives a general overview of the interface.

4.1 Configuration

To use WAMM, users have to write a configuration file which contains the description of the nodes that can be inserted into the virtual machine, i.e. all the machines that users can access and which have PVM installed. This operation only has to be done the first time that WAMM is used.

The configuration file is written in a simple declarative language. An excerpt of a configuration file is shown in the following:

```
WAN italy {
    TITLE "WAN Italy"
    PICT italy.xpm
    MAN cineca 290 190
    MAN pisa 210 280
    LAN caspur 300 430 }

MAN cineca {
    TITLE "Cineca"
    PICT cineca.xpm
    LAN cinsp1 220 370
    LAN cinsp2 220 400
    LAN cinsp3 220 430
    LAN cinsp4 220 460 }

...

MAN pisa {
    TITLE "MAN Pisa"
    PICT pisa.xpm
```

[1] *xpm* is a freely distributable library which simplifies the use of pixmaps in X11; it is available via anonymous FTP at `avahi.inria.fr`

```
    LAN cnuce 200 100
    LAN sns 280 55 }

LAN caspur {
    TITLE "Caspur"
    HOST caspur01
    HOST caspur02
    ...
    HOST caspur08 }

...

HOST cibs {
    ADDRESS cibs.sns.it
    PICT IBM41T.xpm
    ARCH RS6K
    OPTIONS "&"
    XCMD "Xterm" "xterm -sb"
    CMD "Uptime" "uptime"
    CMD "Who" "who"
}

...
```

The file describes the geographical network used, named `italy`. The network consists of some MANs and a LAN. For example, `pisa` MAN includes the local networks `cnuce` and `sns`; `sns` LAN contains various workstations, among which `cibs`.

As can be seen, the network is described following a tree-like structure. The root is the WAN, the geographical network that groups together all the hosts. The children are Metropolitan (MAN) and Local (LAN) networks. A MAN can only contain local networks, whereas LANs contain only the hosts, the leaves of the tree.

Various items can be specified for each declared structure, many of which are optional and have default values. Each node on the tree (network or host) can have a PICT item, which is used to associate a picture to the structure. Typically, geographical maps are used for networks, indicating where the resources are; icons representing the architecture are used for the hosts.

The following is an example of a "rich" description of a host:

```
HOST cibs {
    ADDRESS cibs.sns.it               # host internet address
    PICT IBM500.xpm                   # icon
    ARCH RS6K                         # architecture type
    INFO "RISC 6000 model 580"        # other information
    OPTIONS "& so=pw"                 # PVM options
    XCMD "AIXTerm" "aixterm -sb"
    XCMD "NEdit" "nedit"              # remote commands
    XCMD "XLoad" "xload"
    CMD "Uptime" "uptime"
}
```

In this case the host is an `IBM RISC 6000` workstation; the type of architecture is `RS6K` (the same names as adopted by PVM are used). To insert the node into the PVM, special flags have to be used (`& so=pw` are PVM's own options). The user can execute `aixterm`, `nedit`, `xload` and `uptime` commands directly from the interface on the remote node. For example, using `aixterm` he can connect himself directly to the machine. The `aixterm` program runs on remote node, but the user receives the window on the local terminal and can use it to insert UNIX commands.

4.2 Activation

User starts the interface with the command:

```
wamm <configuration file>
```

PVM does not need to have been already activated. If the virtual machine does not exist at this point, WAMM creates it on the basis of the contents of the configuration file. The "base" window corresponding to the WAN is shown to the user (fig. 2).

4.3 Windows

WAMM visualizes information relating to the networks (at WAN, MAN or LAN level) in separate windows, one for each network. Hosts are shown inside the window of the LAN they belong to.

Figure 2: WAMM, example of initial WAN window

The WAN window is split into three parts (fig. 2). At top left is the map indicated in the configuration file. There is a button for each sub-network; the user can select them to open corresponding windows.

All the hosts declared in the configuration file are listed on the right. The list has various uses: the user can access a host quickly by double clicking on the name of the machine, without having to navigate through the various sub-networks. By selecting a group of hosts, various operations can be invoked by the menu:

- insert hosts into PVM;
- remove hosts from PVM;
- check hosts' status;
- compile on selected hosts.

All the messages produced by WAMM are shown at the bottom. Figure 2 shows information written when the program was started. MAN sub-networks are shown using the same type of window (fig 3). The only difference is in the list of hosts, which, in this case, only includes nodes that belong to the MAN.

For the local networks, the windows are organized differently (see fig. 4). The window reproduces a segment of Ethernet with the related hosts. For each host the following are shown: the icon, the current status (`PVM` means that the node belongs to the virtual machine), the type of architecture and other information specified in the configuration file. Each icon has a popup menu associated with it, which can be activated using the right mouse button. This menu enables users to change the status of the node (add or remove from PVM), run a compilation or execute one of the remote commands indicated in the configuration file. Basic operations on groups of hosts can still be carried out by selecting one or more nodes and invoking the operation from the window menu. In all cases, the results appear in the message area at the bottom of the window.

4.4 Compilation

The compilation of a program is mostly managed by WAMM: the user only has to select hosts where he wants to do the compilation, and call **`Make`**

Figure 3: WAMM, a MAN window representing Pisa

from the `Apps` menu (fig. 2). Using a dialog box, the local directory that contains the source files and the `Makefile` can be specified, along with any parameters needed for the `make` command. No restrictions are made on the type of source files to compile: they can be written in any language.

WAMM carries out some operations needed to compile an application. In the following order:

1. all the source files are grouped into one file, in the `tar` standard format used on UNIX machines;

2. the file produced is compressed using the `compress` command;

3. a PVM task, which deals with the compilation (`PVMMaker`), is spawned on each selected node; the compressed file is sent to all these tasks.

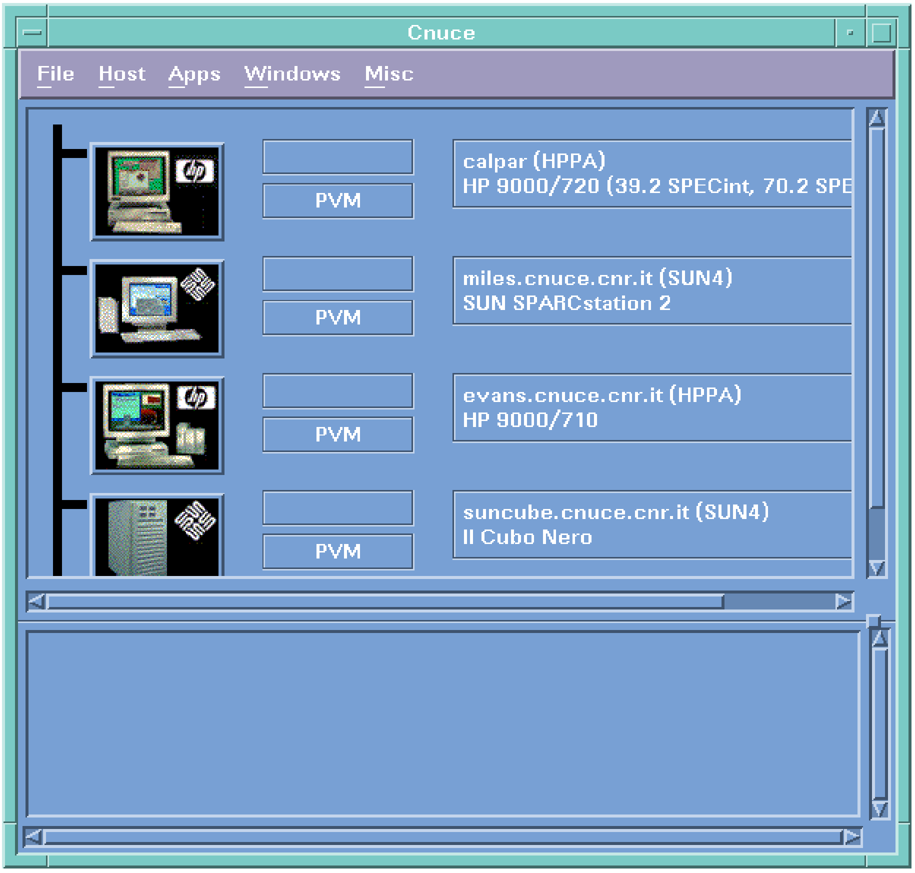

Figure 4: WAMM, LAN window

Now WAMM's work ends: the remaining operations are carried out at the same time by all `PVMMakers`, each on its node. Each `PVMMaker` performs the following actions:

1. it creates a temporary work directory, inside user's home directory;
2. compressed file is received, expanded and saved in the new directory; source files are extracted;
3. the UNIX `make` command is executed.

At the end of the compilation the working directory is not destroyed (but it will be if there is a subsequent compilation). If needed, the user can thus

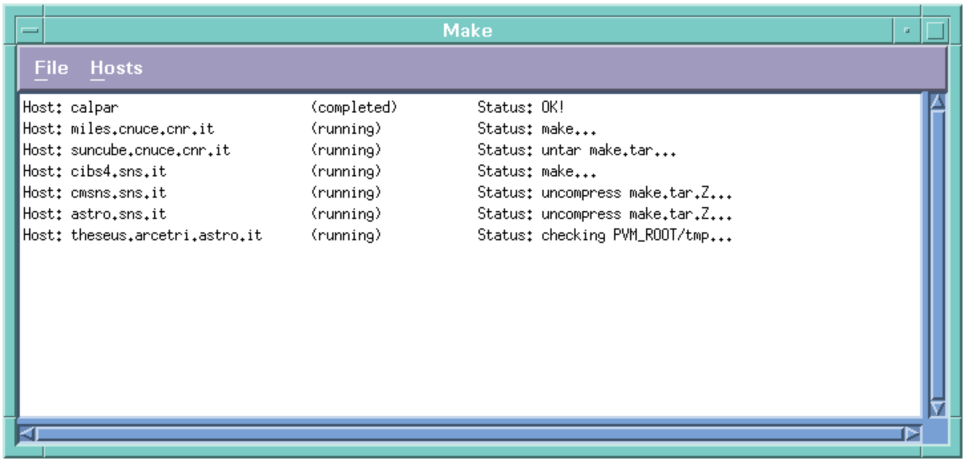

Figure 5: WAMM, control window for the compilation

connect to the host, modify the source code if necessary, and manually start a new compilation on the same files.

Each `PVMMaker` notifies WAMM of all the operations that have been executed. Messages received from `PVMMaker`s are shown in a control window, to let users check how the compilation is going. Figure 5 depicts a sample compilation run on seven machines. For each machine the step that was currently being made when the image was "fixed" can be seen. For example, `astro.sns.it` has received its own copy of the directory and is expanding it, while `calpar` has successfully completed the compilation.

By selecting one or more hosts in the control window, output messages can be seen before the compilation completes, along with any errors caused by `make` and by the compiler. A `make` can be stopped at any moment with a menu command. If a node fails (for example due to errors in the source code), this does not affect the other nodes.

If the compilation is successful, the same `Makefile` that was used to carry out it can copy the executable files produced into the directory

```
$PVM_ROOT/bin/$PVM_ARCH
```

used by PVM as a "storage" for executable files. This operation can be carried out on each node.

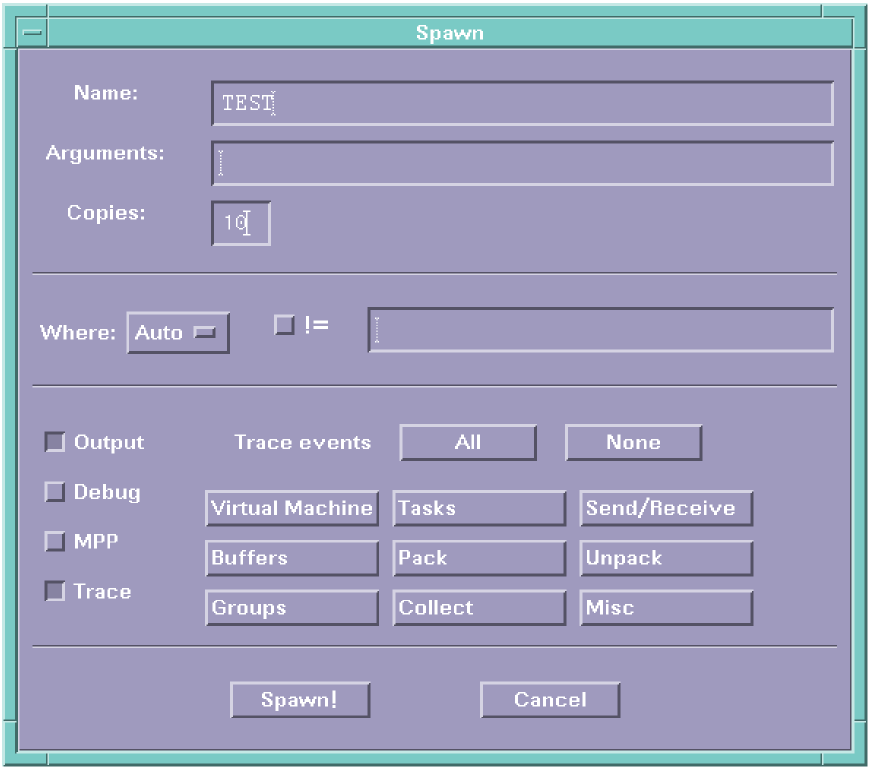

Figure 6: WAMM, PVM process spawning

4.5 Tasks

WAMM allows PVM tasks to be spawned and controlled. Programs are executed by selecting `Spawn` from menu `Apps` (fig. 2). A dialog box is opened where some parameters used by PVM for requesting the execution of the tasks can be inserted (fig. 6). The following can be specified:

- the name of the program;
- any command-line arguments to pass to the program;
- the number of copies to execute;
- the mapping scheme: by specifying `Host` all the tasks are activated on one machine (whose address has to be indicated); by specifying `Arch`,

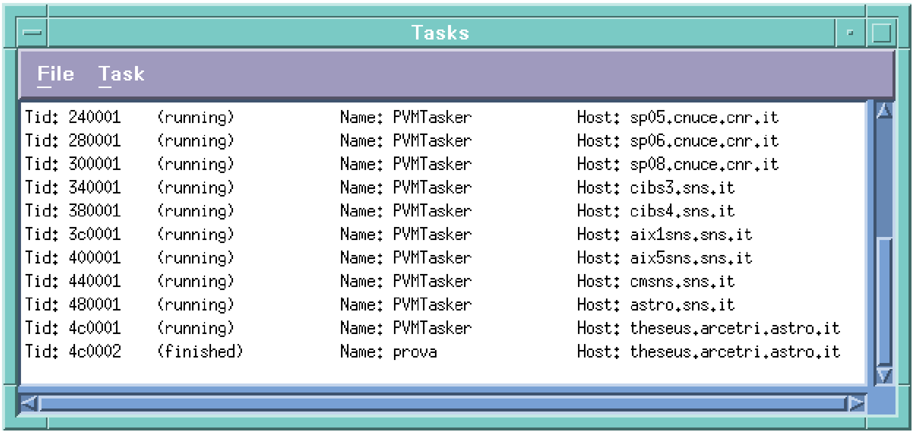

Figure 7: WAMM, control window for PVM tasks

PVM chooses only those machines with a user-selected architecture; finally, `Auto` can be used to let the PVM choose the nodes on which copies have to be spawned;

- PVM's various flags (`Debug`, `Trace`, `MPP`);
- any redirection of the output of the program to WAMM (flag `Output`);
- the events to record if the Trace option is enabled.

By selecting `Spawn`, the new tasks are run. The `Windows` menu (fig. 2) can be used to open a control window that contains some status information on all the PVM tasks being executed in the virtual machine (fig. 7). Data on tasks are automatically updated: if a task terminates, its status is changed. New tasks that appear in the system are added to the list, even if they were not spawned by WAMM. Output of processes activated with the `Output` option can be seen in separate windows and can be also saved to a file. If the output windows are open, new messages from the tasks are shown immediately.

`Kill` (task destruction) and `Signal` (sending specific signal) are possible for all PVM tasks, including those not spawned by WAMM.

5 WAMM's implementation issues

This section outlines the most important aspects of the implementation. Some reference is made to concepts and functionalities of the UNIX operating system and the PVM environment; see [20] and [11] for further details.

5.1 Structure of the program

Each complex function of WAMM is implemented by an independent module; the modules are then linked during the compilation. This type of structure is useful for all complex programs and facilitates modifications to the code and the insertion of new functionalities. The set of modules can be subdivided into three levels:

Application modules. These are high level modules which implement task spawning and control, as well as source codes compilation.

Graphic modules. These include all the functions needed to create the graphic interface of the program.

Network modules. These are control modules which act as an interface between the application and the underlying virtual machine.

The program is totally event-driven. Once the initialization of the internal modules and the related data structures is complete, the program stops and waits for messages from the PVM environment or from the user (for example, the termination of an active task or the selection of a button in a window). This is a typical X11 program behaviour.

5.2 Host control

During initialization WAMM enrolls PVM so that all the control functions of the virtual machine offered by the environment can be exploited. Specifically, the insertion and removal of hosts is controlled using the function `pvm_notify`. WAMM is informed of any changes in the metacomputer configuration and shows them to the user. The notification mechanism is also able to recognize any variations produced by external programs. For example, if hosts are added or removed using the PVM console, the modification is detected by WAMM as well.

5.3 Task control

Unfortunately, PVM's notification mechanisms for tasks is not as complete as that for hosts: by using `pvm_notify` it is possible to find out when a given task terminates, but not when a new task appears in the system. To get complete control of tasks too, WAMM has to use satellite processes, named `PVMTasker`s. During initialization phase a `PVMTasker` is spawned on each node in the virtual machine. Each `PVMTasker` periodically queries its own PVM daemon to get the list of tasks running on the node. When variations from the previous control are found, WAMM is sent the relevant information.

Using `PVMTasker` processes is just one way to emulate a more complete `pvm_notify`. One drawback is that `PVMTasker`s have to be installed on each node indicated in the configuration file. An alternative is to let the interface itself request, from time to time, the complete list of tasks (PVM's function `pvm_tasks` can be used to do this). This method does not need satellites but does have some drawbacks; specifically:

- data from all the daemons have to be transmitted to WAMM, even if there have not been variations in the number of tasks, compared to the previous control. In the first solution messages are only sent when necessary;
- if one of the nodes fails, then `pvm_tasks` waits until a timeout is reached. Some minutes may pass before the function resumes with the next nodes and sends the list of tasks to WAMM. The first solution, which is based on independent tasks, does not have this problem: if a node fails, then only its own tasks will not be updated.

There is a third solution, which exploits PVM's concept of *tasker*[2]. A *tasker* is a PVM program enabled to receive the control messages which, in the virtual machine, are normally used to request the activation of a new process. This basically means that if a *tasker* process is active on a node, the local daemon does not activate the program, but passes the request to the *tasker*. The *tasker* executes it, and when the activated process has terminated, it informs the daemon. We could write a *tasker* in such a way that not only it deals with the daemon, but it also notifies the interface of the activation and the termination of its own tasks. This solution is the

[2] *Taskers*, along with *Hosters*, were introduced with version 3.3 of PVM.

least expensive in terms of communications (periodic messages between the control task and the daemon of its node are eliminated too), but it is not without drawbacks:

- a program still has to be installed on each node;
- no information are available on PVM processes created before the *taskers* are registered.

When developing WAMM we tried out all three solutions. We opted for the first as it was by far the best both in terms of network usage and control capabilities.

5.4 Remote commands

`PVMTasker` processes described above are also used to execute programs on remote hosts: the satellite task receives the name of the program along with command line arguments and executes a `fork`. Child process executes the program; output is sent to the interface. This solution has a main drawback: it is impossible to execute commands on hosts were `PVMTasker` is not running.

The classical alternative consists in using commands such as `rsh` or `rexec`. These can be used for any host, even if it is not in PVM. For example, to find out the load of the node `evans.cnuce.cnr.it`, a user connected to `calpar.cnuce.cnr.it` can write:

```
rsh evans.cnuce.cnr.it uptime
```

The `uptime` command is executed on `evans` and the output is shown on `calpar`. The nodes do not have to belong to the virtual machine (nor, in fact, does PVM have to be installed). The problem arises from the fact that `rsh` and `rexec` can be considered alternatives:

- to use `rsh` the user has to give remote hosts permission to accept the execution requests, by creating an `.rhosts` file on each node used;
- to use `rexec` no `.rhosts` files are needed, but unlike `rsh`, the password of the account on the remote host is requested.

Neither method is really satisfactory: `.rhosts` files create security problems and are often avoided by system administrators; the request for password is not acceptable when there are many accounts or commands to deal with. PVM therefore allows both methods to be used: the user specifies in the hostfile, for each node, what should be used (the same options are admitted in the configuration file used by WAMM). This information is needed since PVM has to activate the `pvmd` daemon on all the remote nodes that are inserted in the virtual machine, by using either `rsh` or `rexec`.

The alternative method to execute a remote command would entail examining the configuration file to establish whether `rsh` or `rexec` is required for each node. In any case it is easier to run the command from the `PVMTasker` on the remote node. With respect to the `PVMTasker`, the command is executed locally, so neither `rsh` nor `rexec` are used.

5.5 Remote compilation

As described in the previous section, WAMM can compile programs on remote hosts by using a PVM task called `PVMMaker`. `PVMMaker`s are spawned, on each node required, only at compilation time and terminate immediately after the conclusion of the `make` process. The compressed directory and the command line arguments for `make` are sent to `PVMMaker`s. Upon receipt of these data, each `PVMMaker` expands the directory, activates the compilation on its own node and sends messages back to the interface about the current activity. Output messages produced by the compilers are also sent to the interface, under the form of normal PVM messages.

Much of what was said for the execution of remote commands is also applicable to the compilation. To transfer a file onto a host the UNIX command `rcp` can be used, but it requires, like `rsh`, a suitable `.rhosts` file on the destination node. The only valid alternative is to use the `ftp` protocol to transfer files, but managing it is considerably more complex than simply transferring data between tasks via PVM primitives.

The execution of commands needed for the compilation could be accomplished by using `rsh` or `rexec` too. However, not only would this lead to the problems described above, but also it would not be as efficient as the solution based on the `PVMMaker`s — the interface would have to manage the results of *all* the operations on *all* the hosts. By using the `PVMMaker`, the interface only has to spawn the tasks, send them the directory with the source files and show the user the messages that come back from the various `PVMMaker`s. The compilation is carried out in parallel on all the nodes.

The disadvantages of using `PVMMaker`s are similar to the ones described for the `PVMTasker`s: a `PVMMaker` has to be installed on each node that the user has access to and compilations cannot be made on a node that is not part of the virtual machine (the `PVMMaker` could not be activated). To resolve the first point, the functionalities of the `PVMTasker` and the `PVMMaker` could perhaps be brought together into one task, thus simplifying the installation of WAMM. However, having two separate tasks offers greater modularity.

6 Future developments

The current version of WAMM is only the starting point to build a complete metacomputing environment, based on PVM. Some implementation choices, such as subdividing modules with a similar structure, were made in order to simplify as far as possible the insertion of new features. Possible new features are shown in the following.

Resource management. The management of the resources of a metacomputer (nodes, networks, etc.) is one of the most important aspects of each metacomputing environment. WAMM currently let users work only with the simple mechanisms implemented in PVM to control the nodes and task execution. For example, to spawn a task, the user can only choose whether to use a particular host, a class of architectures or any node identified by PVM on a round-robin basis. For complex applications, more sophisticated mapping algorithms are needed. Such algorithms could be implemented in the module that activates the tasks.

Performance analysis. The current version of WAMM allows the activation of a task in trace mode: whenever the task calls a PVM function, a trace message is sent to the interface. By appropriately recording and organizing these data, all the information needed to study program performance can be obtained. Specifically, times spent on calculation, communications and various overheads can be determined. At the moment WAMM does not make any use of the trace messages it receives; a future version should have a module which can collect these data, show them to the user either as graphics or tables, and save them in a "standard" format, which can be used in subsequent examinations via external tools, such as ParaGraph [15].

Remote commands. The possibility to execute remote commands could be exploited to run the same operation on several hosts simultaneously. This type of functionality, not implemented yet, would allow some problems regarding the development and maintenance of PVM programs to be resolved. For example, with one command an old PVM executable file could be deleted from a group of nodes. Manual deletion is not feasible if there are many nodes, it would therefore be advantageous to use the interface.

7 Related works

WAMM's features can be divided in two groups: first, it provides users with a set of facilities to control and configure the metacomputer; on the other side, it can be also considered as a software development tool. There are some other packages which offer similar functionalities.

XPVM is a graphical console for PVM with support for virtual machine and process management. The user can change metacomputer configuration (by adding or removing nodes) and spawn tasks, in a way similar to WAMM's. With respect to WAMM, XPVM does not provide the same "geographical" view of the virtual machine and is probably more suitable for smaller systems. XPVM has no facilities for source file distribution, parallel compilation and execution of commands on remote nodes; anyway, it includes a section for trace data analysis and visualization, not implemented in WAMM yet.

HeNCE [13, 14] is a PVM based metacomputing environment which greatly simplifies software development cycle. In particular, it implements a system for source file distribution and compilation on remote nodes, similar to that used by WAMM: source files can be compiled in parallel on several machines and this task is controlled by processes comparable with `PVMTasker`s. HeNCE lacks all the virtual machine management facilities provided by WAMM; for these, it is often necessary to use PVM console. It should be said that HeNCE was designed with different goals: the simplification of application development is mostly achieved by using a different programming model, with communication abstraction, rather than providing remote compilation facilities.

References

[1] L. Smarr, C.E. Catlett. Metacomputing. Communications of the ACM, June 1992, Vol. 35, No. 6 (45- 52).

[2] R. Freund, D. Conwell. Superconcurrency: A Form of Distributed Heterogeneous Supercomputing. Supercomputing Review, Oct. 1990, pp. 47-50.

[3] A.A. Khokhar, V.K. Prasanna, M.E. Shaaban, C. Wang. Heterogeneous Computing: Challenges and Opportunities. Computer - IEEE (18-27), June 1993.

[4] P. Messina. Parallel and Distributed Computing at Caltech. Technical Report CCSF-10-91, Caltech Concurrent Supercomputing Facilities, California Institute of Technology, USA, October 1991.

[5] P. Messina. Parallel Computing in USA. Technical Report CCSF-11-91, Caltech Concurrent Supercomputing Facilities, California Institute of Technology, USA, October 1991.

[6] P. Huish (Editor). European Meta Computing Utilising Integrated Broadband Communication. EU Project Number B2010, Technical Report.

[7] P. Arbenz, H.P. Luthi, J.E. Mertz, W. Scott. Applied Distributed Supercomputing in Homogeneous Networks. IPS Research Report N.91-18, ETH Zurich.

[8] V.S. Sunderam. PVM: a Framework for Parallel Distributed Computing. Concurrency: Practice and Experience, 2(4):315–339, December 1990.

[9] G. A. Geist, V. S. Sunderam. Network-Based Concurrent Computing on the PVM System. Concurrency: Practice and Experience - Vol. 4(4) - July 1992.

[10] J. J. Dongarra, G. A. Geist, R. Manchek, V.S. Sunderam. The PVM Concurrent System: Evolution, Experiences and Trends. Parallel Computing 20(1994) 531-545.

[11] A. L. Beguelin, J. J. Dongarra, G. A. Geist, R. Mancheck, V. S. Sunderam, and W. Jiang. PVM3 users' guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Lab, May 1994.

[12] R. Baraglia, G. Bartoli, D. Laforenza, A. Mei. Network Computing: definizione e uso di una propria macchina virtuale parallela mediante PVM. CNUCE, Rapporto Interno C94-05, Gennaio 1994.

[13] A. Beguelin, J. J. Dongarra, G. A. Geist, R. Manchek, V.S. Sunderam. Graphical Development Tools for Network-Based Concurrent Supercomputing. Proceedings of Supercomputing 91, Albuquerque 1991.

[14] A. Beguelin, J. J. Dongarra, G. A. Geist, R. Manchek, K. Moore, R. Wade, J. Plank, V.S. Sunderam. HeNCE: A Users' Guide, Version 2.0.

[15] M. T. Heath and J. E. Finger. ParaGraph: a tool for visualizing performance of parallel programs. Oak Ridge National Lab, Oak Ridge, TN, 1994.

[16] G. Bertin, M. Stiavelli. Reports on Progress in Physics, 56, 493, 1993.

[17] S. Aarseth. Multiple Timescales. Ed. J.U. Brackbill & B.I. Cohen, p.377, Orlando: Academic Press, 1985.

[18] L. Hernquist. Computer Physics Communications, 48, 107, 1988.

[19] H. E. Bal, J. G. Steiner, A. S. Tanenbaum. Programming Languaues for Distributed Computing Systems. ACM Computing Surveys, Vol. 21, No. 3, September 1989.

[20] H. Hahn. A Student's Guide to UNIX. Mc Graw-Hill, Inc., 1993.

[21] M. Brain. Motif Programming: The Essentials ... and More. Digital Press, 1992.