Massive Parallelism with Workstation Clusters – Challenge or Nonsense?

Technical Report IFI-TR 94.01

CLEMENS H. CAP*
Department of Computer Science
University of Zurich
Winterthurerstrasse 190
CH–8057 Zurich, Switzerland
e-mail: cap@ifi.unizh.ch, Tel. +41–1–257 43 26

December 14, 1993

Abstract

Workstation cluster computing recently has become an important and successful technique. The communication bottleneck limits this approach to small and medium sized configurations of up to 30 workstations for most applications. This paper demonstrates that for certain algorithms massively parallel cluster computing using thousands of workstations in the Internet is feasible. It describes structures for the coordination of a large number of geographically dispersed processes. The paper introduces LOLA, a library supporting massively parallel computing in wide area networks, and provides an application example.

1 Introduction

Recent research activities have shown, that workstation cluster computing or *hypercomputing* provides a fair amount of computing power at a reasonable price. [16] reports a performance of more than 1 Gflops in a cluster of 110 workstations and [23] demonstrates, that for certain applications cluster solutions provide a price performance ratio which is two orders of magnitude better than for classical supercomputers. [32] estimates that

*Research partially supported by Firma Siemens AG München and Schweizer Bundesamt für Konjunktur und Wirtschaftsförderung, grant numbers 2255.1 and 2554.1.

90% of Lawrence Livermore National Laboratory's workload could be solved with workstation clusters in reasonable time. On the other hand, [10] shows, that communication represents a severe bottleneck of a hypercomputer. One might also expect a limit on the number of workstations, since usually a larger number of processors also means an increased communication load.

At first sight the idea of massive parallelism using workstations interconnected by local and wide area networks seems to make no sense, especially if we use the term massive parallelism for configurations with at least a three or four digit number of processors. This paper will demonstrate in Section 2, that for certain important application problems, massively parallel cluster computation in a WAN indeed is an option and a challenge, whereas for a general approach it makes no sense. Section 3 describes techniques for the development of massively parallel applications for wide area networks and Section 4 describes concrete library routines for this purpose. Section 5 discusses practical experiences gained from implementations and applications. The paper closes with an overview of similar research activities and future work.

2 Massive Parallelism in WANs – Challenge or Nonsense?

2.1 Is Massive Parallelism in a WAN Possible at All?

Computations simulating two-dimensional heat conduction using a cluster of 40 Sparc workstations produced an average communication bandwidth of 5.288 MBit/sec or 52.88 % of the physical bandwidth of an Ethernet, under the assumption that communication takes place during the entire run [11], [10]. The peak communication load therefore was still higher. An Ethernet utilization of more than 20 % is already considered a high load. These experiments and analytic performance models indicate, that the communication bandwidth limits the scalability of this algorithm, since the communication load is proportional to the number of processors. Furthermore the high number of collisions in a heavily loaded Ethernet has an adverse effect on speedup. Similar observations can be made for many applications due to their small grain size and locality [4], [9].

There can be at most 255 workstations in a local area network, and on the average there are only some 20 machines in a LAN. Therefore massive parallelism with workstation clusters will have to use several LANs connected by metropolitan and wide area network technology. With the present communication technology the following sustained bandwidths were observed in Internet connections [21]:

National (Switzerland)	100 kBytes/sec
Transnational (Europe)	1 – 50 kBytes/sec
Transatlantic	$\leq$ 30 kBytes/sec
Transpacific	$\leq$ 3 kBytes/sec

Therefore only special applications and algorithms with *low communication* demands are reasonable for massively parallel cluster solutions. Depending whether the communication of two processes takes place within a LAN or crosses the borders of LANs, differences in the bandwidth of two or more orders of magnitude may be observed.

There are a number of applications, where the *dominating operations* are comparisons, jumps and integer operations. Their programs are much better adapted to the architecture of a RISC workstation with their relatively fast scalar integer units than to the traditional supercomputer with extremely fast floating point units but comparatively slow scalar integer units. For example the BLASTPM code from molecular biology required 1820 sec. computing time on a Sun 4 workstation and 1522 sec. on a CRAY Y-MP [18]. Similar experiences are reported by [23].

Many problems from *mathematical physics* are concerned with a local physical law, which is given by a differential equation. Their dominating operations are floating point operations and the locality of the physical law is reflected in the algorithm by a high demand for interprocess communication. However, a large class of problems from optimization and simulation, from information retrieval, from VLSI layout to the analysis of homology in genetic sequences require only a very small amount of communication and allow a rather large degree of parallelism at the same time. The fact that these algorithms often are not in the central interest of the high performance computing community might go hand in hand with the unsuitability of these problems for classical supercomputer architectures.

Consequence: Massive parallelism using workstations in a wide area network is possible, but only for special applications and algorithms.

2.2 Do We Need Special Control Mechanisms in a WAN?

There presently exist many software packages which assist the programmer in developing parallel programs for workstation clusters [9], [32]. In principle these systems could also control parallelism within wide area networks, because they are independent of the communication medium as long as a suitable interfaces, like the Berkeley socket abstraction, are available.

For practical purposes *wide area communication is different* from the local area situation for a variety of reasons: The bandwidth in a WAN is one to two orders of magnitude lower than in a LAN and latency is higher by one order of magnitude. In a WAN there can be much additional traffic from other interconnected LANs and so there is a higher variation in latency and bandwidth. Commands which start and terminate processes on remote machines are likely to be delayed for several seconds or even minutes. The view of a central coordinating site from remote machines therefore is outdated by the same amount of time. Time out mechanisms which signal that a workstation or network connection is not available have delays of up to several minutes.

Although *location transparency* of processes in a cluster is desirable from a software engineering point of view, in wide area massively parallel systems the programmer should know, whether two processes reside in the same LAN or not. Communication and synchronization of processes is much more difficult if it must be achieved across the borders of a LAN and should therefore be adapted to the topology-in-the-large of the wide area network.

Small clusters of 5 to 20 machines often belong to one department and may be used in dedicated mode for experimentation. When using several hundred workstations there will always be some activity by other users who are likely to revoke the permission for using their machine if they are disturbed in their own work. Therefore we must take special care not to produce too much load when the local user needs his workstation. Furthermore all problems associated with looping programs, filling the file system with core dumps, consuming all slots of the process table, using up temporary storage space, flooding local networks, producing extensive swapping activities lead to much more trouble if they occur on thousand machines in a WAN than on ten machines in your own LAN.

Due to the distributed ownership of a wide area massively parallel cluster we have less *influence on the configuration* of the machines. In general, superuser capabilities for installing programs or reconfiguring systems are available only for our own workstations. Therefore we must cope with heterogeneous configurations, which cannot be flexibly adapted to our needs. For example, temporary storage space could be found in different parts of the directory structure, network services like name servers could be configured differently or are even disabled.

Since we have to work with a large number of different hardware architectures and operating system releases, a *highly portable* programming style is necessary. The *installation* of our software cannot be made by hand on several hundred machines. Portable shell scripts must be developed and configured using parameter files to automate this task. In a wide area environment we have less control on remote equipment and a much higher possibility of workstation or network *failure*. Especially, a workstation might be turned off unexpectedly by its local user, who is not informed about the parallel processes on his machine.

Consequence: All cited effects provide a unique challenge for controlling communication and process activities in a wide area domain and necessitate distinguishing an *inter-LAN* and an *intra-LAN* layer.

3 Managing Massive Parallelism in a WAN

We use three types of processes for controlling the activities of our cluster. *Worker* processes perform the computational work of the application; all the other processes are mainly responsible for the distributed coordination of activities. A single *master* process takes care of all top level system activities, provides a user interface and starts slave and worker processes. The *slave* processes are necessary to coordinate larger configurations. Coordinating several hundred worker processes from a single master would produce a bottleneck at the master and require the communication of status information from all the workers to the master across the entire network. Therefore the master starts slave processes, preferably a small number in every LAN, and the slave processes administrate several dozens of worker processes. However, a master process may also directly start a worker process and a slave process may start other slave processes, and so the construction of arbitrary tree-structured control hierarchies is possible. Figure 1 shows an example of such a control structure. This tree is determined by a set of host files, one for each of the slaves and for the master. It contains the information, which binary the master or the respective slave has to start on which machine.

The *communication topology* on the inter-LAN layer is a tree with the master process as root, the worker processes as leafs and the slave processes in between. The worker processes within a LAN may use additional means of communication to exchange data among them. Since there are several cluster software systems like Parform [10], PVM [15], P4 [6] and many others [9] which support this kind of communication we will not further elaborate upon this. Communication on the inter-LAN layer can go *upstream* (up the tree towards the root) or *downstream* (down the tree towards the leafs) and is under the control of the wide area cluster software.

Workers from different LANs, or more precisely, workers controlled by different slaves, cannot communicate directly with each other but are required to route their information upstream using the lowest common ancestor in the tree. At first sight this may seem to be a drawback. However, in practice the structure of the tree will be adapted to the physical layout of the wide area network. In this case a direct, point-to-point connection between two workers from different LANs requires the assistance of routing software. In our solution this fact only becomes more explicit. Furthermore we made the observation that many networks are configured in a manner that only one or two servers have direct access to the outer world network for performance and security reasons. In such a situation a direct network connection cannot be established and additional routing processes must be developed. This problem is circumvented by placing the slave processes on those

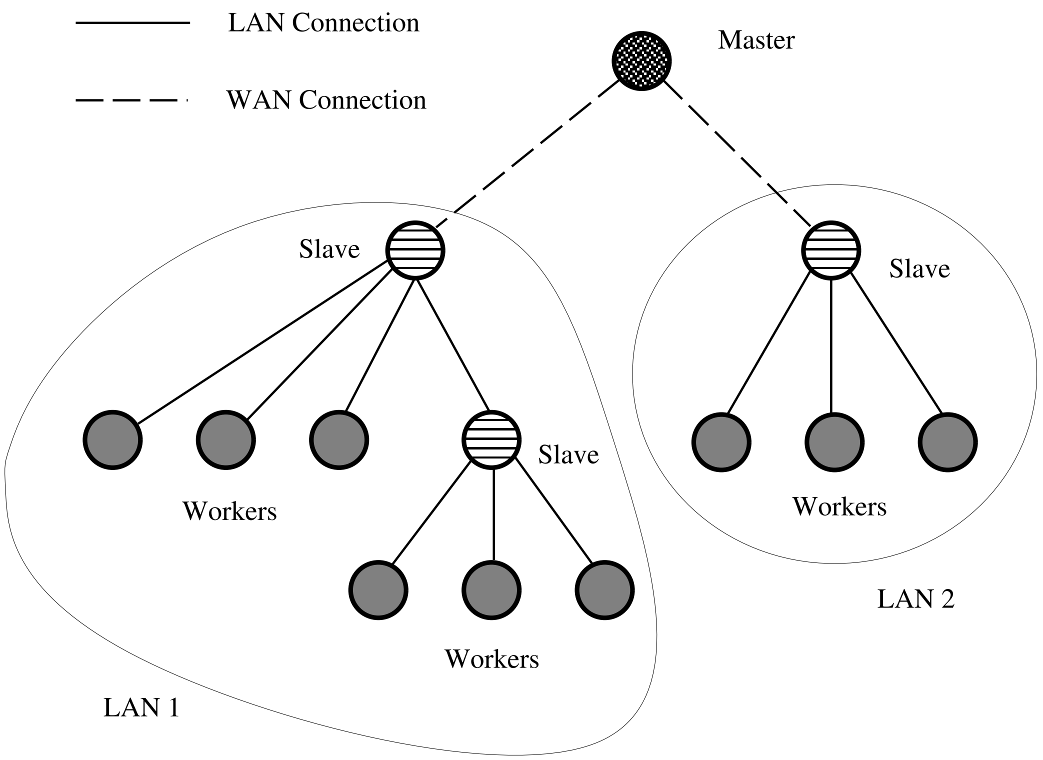

Fig. 1

machines which have direct network access. Then data packets between worker processes on different LANs automatically are routed by these slaves.

The processes themselves are identified by a *hierarchical name space* which additionally provides routing information, where necessary. The master process is identified by the empty routing path **()**, the process started by the master are identified by **(0)**, **(1)**, **(2)**, etc, the processes started by the slave **(1)** are identified by **(1/0)**, **(1/1)** and so on.

Before an application can be run, all necessary processes must be *started up*. When the master is started, it consults its host file to find out which processes must be started on which machine. Once a slave process is started on a remote machine, this process repeats itself and after a while the entire tree of master, slave and worker processes is operative.

As we outlined already, it is imperative to prevent any disturbance of other users. Therefore we must provide an elaborate automatic *shutdown mechanism* which is triggered on a number of occasions. First, every operating system call requires full error handling. A large number of error conditions like "file system full" or "process table full" and system related signals like "segmentation fault" must be observed and caught. Then all available error information is sent upstream, so that the reason of the shutdown can be identified by the user on the machine running the master process. Next, a cleanup procedure is called for deleting temporary storage files and for killing all other processes running under the same user identification. At last the process exits itself. If the process where this shutdown mechanism is triggered is a slave, then the communication connection to its children in the tree is lost. When these child processes try to communicate with the slave, they will receive an error condition and trigger the shut down process themselves. It is therefore guaranteed, that the shutdown of a non-leaf process eventually also leads to a cleanup and shutdown of all children in the tree and no processes will be left over. Implementing a periodic heartbeat "are you still there" enquiry, running from the root to the leafs, ensures, that this shutdown will be performed within a predetermined period of time. Furthermore, all processes implement a watchdog mechanism. If the watchdog reset function has not been called for, say, more than one minute, then the watchdog triggers the shutdown procedure.

Once a process is started it receives information, where and how it can establish communication with its parent. Then it starts its own child processes and waits a certain amount of time for status information from these children. All children that did not provide status information during this period are considered in error. Then the slave composes a concise summary of the received status information which it sends upstream to its own parent process. Now the slave waits for application commands from upstream and exits, unless a command is received in due time.

This mechanism produces a downstream *wave front* running from the root of the tree to its leafs: The master starts some slaves, they again start slaves and workers and so on until all processes are started. Then the leaf processes start a tiny benchmark program to determine the computational performance with which they might contribute. This information forms an upstream wave front which is running from the leafs to the root. At all branches of the tree a summary of all the information received from downstream is composed and sent upstream, error information is passed from downstream to upstream without any modification for display to the user. Having received the summary information which finally reaches the master, we supply the master process with application data. This information is broadcast downstream again, using a similar mechanism. When this information finally reaches the leafs, the worker processes start their application calculation. When they are finished, the result is sent upstream again. The slaves apply an application dependent function to the data they receive from their children and pass the result further upstream. The final result is output by the master process.

When using a larger configuration we naturally cannot predict, which workstation will be running and which will be turned off. Therefore the application algorithm used for the worker processes must be able to deal with a varying number of workers and even provide for the unexpected loss of a worker during the calculation.

This can be done by implementing a LINDA-like *workpool*, which is administrated by the slave which is the immediate common ancestor of the workers: The slave keeps track of the necessary parts of the work. It hands out work to all available workers and receives the results of the computation. It periodically checks, whether there are new workstations, which can be used for the computation and keeps track, whether for work handed out eventually a result is returned. If a worker crashes, the slave hands the respective part of the work to another worker. When a worker crashes all its state is lost. Therefore the slave-worker cooperation must be modeled as a stateless client-server cooperation. If a work parcel only represents an intermediate step of the calculation the result which is returned by the worker must contain all information to enable a different process the continuation of the computation. An efficient form of this workpool usually is highly application dependent and hence is implemented by the application programmer.

At the inter-LAN layer no fault tolerance mechanism is used. Since the slave processes are running on servers or gateways with a connection to the WAN, they generally are more stable than the individual workstations and will not be turned off unexpectedly. The slaves usually produce only negligible computational load and so there is no danger of overloading a server.

The *inter-LAN programming model* is based on waves which traverse the tree and provides tree structured communication at low bandwidth and high latency time. Communication between arbitrary processes is difficult to establish for processes in different LANs. The *intra-LAN programming model* uses the workpool mechanism and can handle every communication topology. It provides a significantly higher bandwidth and smaller latency time. Application algorithms must be adapted to exploit these structures.

Since the activation of a master or slave process starts a large number of processes all over the place it is advisable to implement a global *password mechanism* in addition to UNIX password security, in order to prevent unauthorized persons from accidentally starting these programs. This password is typed in upon startup of the master and is broadcast to all other processes in the sequel. It can conveniently be stored in a UNIX-like password file. Some of the process startup mechanisms require the clear text password of the corresponding UNIX account. If the password on all accounts is identical it can also be typed in and broadcast. A more flexible solution allowing differing passwords hardcodes the password into program text. This part of the program then is separately compiled and the program text is deleted afterwards. This password module can be linked to the application and provides the clear text password upon request of the account owner or a cooperating program. Care must be taken to construct the password string by `char` assignments instead of string assignments because entire strings can be retrieved from object code using the UNIX command `strings`. Theoretically this file could be disassembled or linked to programs with a fake protocol to retrieve the clear text password. If correct file permissions are used, then this at least requires unobserved access to the account and sufficient knowledge for disassemblation.

In order *not to disturb the local user* of a workstation by placing a production job onto his machine, a number of techniques may be used. In the host file we can specify the scheduler priority (niceness) for all processes. A sufficiently low priority guarantees, that our process gets a time slice only, when no other process is runnable. This technique works fine only, when our process consumes just a small percentage of the workstation's RAM; otherwise each time the process gets a time slice or must relinquish the CPU to another process, some time must be spent for swapping and paging. A user working with an interactive program is likely to be annoyed by that additional delay. In this case one could program the workpool of the slaves in a way, that a machine only gets a piece of work, if it has been idle for at least ten minutes. In case the user unexpectedly returns, the workstation signals the slave that it is no longer available and that it will not finish the assigned task. The workpool mechanism will then dispatch this piece of work to a different computer.

For the detection of errors and the control of process behaviour we need *logging facilities*. We should be able to define different levels of logging, since during program development extensive debugging information should be produced, which during production runs should not reduce execution speed. Below a small part of such a log is given, illustrating several successful startups and a few error conditions. The last line informs us, that the startup was able to start 616 worker processes for which the tiny benchmark measured 21727.1 Mips of which 20245.6 had been available to the benchmark, the remaining performance having been consumed by other processes. This resulted on the average in 35.3 Mips per process of which on the average 32.9 were available for the benchmark. In total 92.0% percent of all time slices were available for the benchmark. The unit Mips is measured in a tiny loop and is scaled to yield 40 Mips for a 40 Mhz Sparcstation 2.

```
***ILLEGAL MESSAGE FORMAT***, initial bytes 83/101/103, detected by master (),
sent by child process 31 (147.52.128.1), CONTENTS IS "Segmentation fault (core dumped)"
risc1    (27/)  TOTALS 11,     570.0Mips     569.4TMips     51.8avg  51.8tavg  99.9%
ibk      (38/)  TOTALS 7,      411.9Mips     383.8TMips     58.8avg  54.8tavg  93.1%
dur      (41/)  TOTALS 1,      45.2Mips      26.0TMips      45.2avg  26.0tavg  57.6%
berg1    (24/)  TOTALS 11,     481.3Mips     407.1TMips     43.8avg  37.0tavg  84.7%
***ILLEGAL MESSAGE FORMAT***, initial bytes 66/97/100, detected by master (),
sent by child process 33 (129.194.77.2), CONTENTS IS "Bad system call"
nem2 (30/) ***ERROR 25*** detected on child processor 139.91.10.19 archimedes.csi.forth.gr,
error codes are 203 0 -1 0.0 0.0 0 0.0
master   ()     OVERALL 616,   21727.1Mips   20245.6TMips   35.3avg  32.9tavg  92.0%
```

If we want to fine tune the performance of our system, a *load monitoring facility* is necessary. Our group recently developed a tool for visualizing the load situation of a very large number of workstations situated in different LANs.

4 Library Routines

As *programming language interface* to the described techniques a small set of library routines is provided, which can be used to compose master, slave and worker processes. Master and slaves usually are small programs which can be derived by adapting a template to the specific application, the workers contain the main application code. In the following we will describe some of the routines.

`setup(mode)` configures a process as a master (mode = 0), slave (mode = 1) or worker (mode = 2) process. This is necessary to adapt the library routines to the specialities found in these processes and allows the use of routines with identical names for similar tasks in all three types of processes.

Processes on remote workstations can be started using a variety of techniques. The parent process can contact the Internet-demon `inetd`, issue the remote shell command `rsh`, or it can contact the remote shell server `rshd` or the remote execution server `rexecd` directly. Since several sites disabled some of these network services such a large choice of methods was necessary. We even worked with a site which had all services disabled, which could remotely start processes. In this situation a user owned and modified Internet-demon of restricted functionality was used instead of the root owned `inetd`. This demon had to be started manually if necessary, or on a regular basis using `at` or `cron` commands. These remotely started processes then must reconnect to their parent, i.e. they must establish a communication link with their parent process. Several employed startup techniques, like the `inetd` already furnish such a connection, for others, like the `rsh` technique they must be established separately. For every machine a suitable startup and reconnect technique is specified in the host file.

If a machine, on which a process shall be started remotely, is not running, it may take several seconds up to one minute until a time out and a negative acknowledgement is generated by the operating system. Therefore it is necessary to parallelize the process startup; eight workstations which are switched off could otherwise increase total startup time to more than ten minutes. While we are waiting for an answer from one workstation we can already initiate the startup sequence on other machines. In case of the `inetd` and the other client-server oriented techniques nonblocking connection attempts were used. The `rsh` technique may be parallelized running several shells with the background operator (&) of the command interpreter. In this case care must be taken to disconnect standard input, output and error files from the remote shells to allow an exit of the remote shell before the exit of the issued remote command. Otherwise every remotely started process would consume one slot in the process table of the slave machine, disabling sooner or later the startup of any additional process on this workstation. The routines `startup_kids()` and `reconnect()` are responsible for these startup and reconnection mechanisms.

`setup_lola()` performs various initializations, prohibits writing of core dump files by generating an empty core file with file permissions which deny write access even for the owner of the process. It sets up signal handlers to catch error conditions and sets up the watchdog.

`receive_upstream (buffer, bufflen)` receives a message from the parent process, `receive_downstream (child_id, buffer, bufflen)` receives a message from the specified child, `send_upstream (buffer, bufflen)` sends a message to the parent process and `send_downstream (child_id, buffer, bufflen)` sends a message to the specified child. `flood_downstream (buffer, bufflen)` broadcasts a message to all children. All receive functions are implemented as blocking and all communication functions perform full error handling. They trigger the shutdown mechanism whenever this is appropriate.

`pass_info (app_dep, a, b)` provides the main synchronization function for the wave mechanism and waits for messages from all the children of the process wherein it is called. As soon as a message arrives it is read and its type and source is decoded from the header. "Pass-through" messages are forwarded upstream to the parent, a technique mainly used for error messages which the processes traditionally would write to the standard error file. "Status" messages update a table reflecting the state of the immediate children. "Finished" messages trigger the release of the synchronization mechanism and are further described below. "Application" messages call an application dependent function for further processing. This function is specified by the pointer `app_dep` and is passed the contents of the message for further processing. The thread of control finally returns to `pass_info`. Control is released from `pass_info` after a timeout, specified by the parameters `a` and `b`, occurs, or after every fully operative child process has sent a "finished" message. This mechanism allows to leave `pass_info` earlier than the specified time out and as soon as the children signaled, that they have sent all necessary information. In case an insufficient number of finished messages is received, this indicates a possible problem with one of the children and starts code which detects, whether all children are still operative; if appropriate, the shutdown mechanism is triggered.

`send_finished()` sends a "finished" message upstream and `send_state_upstream()` sends state information on the process (for workers) or statistical summaries (for slaves) upstream. Both messages are automatically dealt with in `pass_info`.

The master and all slaves collect *state information* on their child processes. Among others they record the number of operative and erroneous processes, the performance of the workstations, determined by a tiny artificial benchmark, the percentage of this performance, that could be contributed to the parallel run and information about the presence and activities of other processes and of interactive users. Every process composes a concise statistical summary, which is sent to the parent process.

`cleanup()` performs duties like clearing all files we wrote into temporary storage, sends kill signals to all other processes of our user id, removes core dump files and prepares a clean shut down of the process.

`watchdog()` resets the watchdog. If the watchdog is not reset for a period of time to be specified in a parameter file, the shutdown sequence is initiated.

`log_file (filename)` specifies a file for local logging. `start_log()` and `stop_log()` turn logging on or off by I/O redirection. Various levels of logging can be chosen by setting preprocessor variables before the compilation of the library. In the highest level of logging a very large number of messages is generated all over the program. A dynamic declaration of logging levels would require conditional statements for every logging command and would lead to an increased code size and a decreased execution speed, which is disadvantageous in production runs.

5 Implementation and Practical Experiences

To test our approach, above routines were *implemented* in C for 23 different workstation/operating system combinations and installed on over 800 workstations from up to 31 local area networks of the Internet, situated in all five continents of the globe [12], [8]. The system is nicknamed LOLA, which is an abbreviation for <u>lo</u>osely coupled <u>l</u>arge <u>a</u>rea and was used to develop a prototype for the comparison of genetic sequences.

Portability for the various systems was astonishingly easy to achieve by a small number of preprocessor directives. The problems encountered were mainly associated with different paths for include and header files and with a small number of system calls like `getrusage`, which only were available on UNIX brands closely related to the Berkeley system. On some System 5 releases and on Solaris 2 environments compatibility libraries had to be linked. In general portability was no big issue despite the heavy use of low level network interface routines.

For the *installation* accounts were set up on the remote machines and a master site was entered into the `.rhosts` file. A set of shell scripts was developed for automating the installation: At first various scripts were transferred to the remote hosts via `rcp`. The copied scripts then configured, compiled and installed the software on the target machines. In order to deal with various different configurations encountered during the installation we developed a parameter file whose entries allowed a detailed control of the various installation options. In *heterogeneous* networks we had to ensure, that every machine executes the binary suitable for its architecture. This was especially difficult, when only one machine was connected to the WAN and two step copying was necessary to provide workstations of a different architecture with their sources. In networks with *homogeneous* hardware we encountered different and incompatible versions of runtime systems and shared libraries, which led to error conditions. On many occasions only "version mismatch" messages were generated, but also more severe incompatibilities could be found. This problem was solved using static linking of libraries in the compilation process. Due to the size and heterogeneity of the configuration manual installation via `ftp` or `telnet` or WAN-wide network file systems were no option. However, the necessity of the shell scripts and their functionality became obvious only during our experimentation and so many installations initially had to be done manually.

The *incoherent configuration* we found in a number of networks was disturbing, and even worse, well known *security holes* which had not been closed by systems administration gave an alarming picture of the state of some academic workstation environments. Fortunately many, especially European departments, gave us a helping hand when problems occurred and assisted us during the installation phase wherever they could. We also could point out some of the security holes we found.

We tested our system with an *application from molecular biology*. Given a new gene sequence the task is to determine all those genes of a database which are most similar to the given one. This application is very important in modern gene technology and its background is described further in [19]. The employed algorithm was developed by Smith and Waterman and uses dynamic programming techniques. It is further explained in [29] and [33]. For parallelization we partitioned the database consisting of 106.684 genes into subsets which were distributed to the individual LANs. The new, given gene was entered at the master process and broadcast to the slaves and worker processes. The slaves told the worker processes to calculate similarity scores between certain parts of the database and the given gene, using the similarity scores of Smith-Waterman. This handing out of work was implemented by a workpool mechanism. Finally, the obtained similarity scores were communicated upstream. Every slave determined the best matching genes, so that finally a small set of best fitting genes was output at the master process. The database of a total size of over 100 MBytes was predistributed to the LANs based on a priori performance estimations. Within a single LAN the slaves and the workpool mechanism provided for load balancing between the worker processes. Only the new, given gene and the obtained scores and gene identifiers had to be communicated, so there was no communication bottleneck. The table below shows the speedup obtained in different experiments [12], [8].

Number of Workstations	Parallel Runtime	Runtime on one Sparcstation 1	Runtime on one Sparcstation 2	Speedup vs. Sparcstation 1	Speedup vs. Sparcstation 2
642	3.2 min	16.4 hours	—	311	—
803	5.0 min	21.5 hours	10.4 hours	258	125
803	18.4 min	6.6 days	3.2 days	515	250

Since this experiment was conducted as *feasibility study*, no effort was made to tune the configuration for high performance. Although in some instances the first local area networks were idling already after a fifth of the total execution time no further load balancing between the LANs was made, and precautions were taken, not to overload the machines and communication links. From this point of view the obtained results are promising. The stability observed throughout the experiments also was astonishingly high.

There are three areas, where the *application* of wide area massively parallel systems is of interest. First, applications which allow a very large degree of parallelism and are well adapted to workstation architectures, similar to the cited example from molecular biology. Second, systems which are WAN based by their infrastructure and data distribution. Examples are name service systems or wide area library catalogues. Here the emphasis is placed on the distributed aspect but we can benefit from speedup effects, too. A third application might be massive parallelism on the supercomputer level. Here the main problem is an imbalanced computation to communication ratio.

It should be possible to setup a WAN massively parallel production system for closed user groups, for example biologists, even across the borders of continents. Small adaptions of network configurations are necessary, in order to obtain higher efficiency. For example, a sufficient number of machines where the slave processes will run, should have network connection, and at least one disc partition should be mounted by all machines of a LAN for storing shared and temporary data. A usage policy must be developed and should be acceptable for all workstation owners. For example, parallel runs should be restricted to certain times of the day (or rather, of the night or of the weekend) and scheduled to machines, whose performance is not required locally.

6 Similar Work and Further Research

WAN parallelism has been studied in [27], [21], [30]. In their experiments a smaller number of *supercomputers* was coupled for optimizing the structure of large molecules with a modified Hartree–Fock method [21]. Massively parallel approaches were studied with *110 workstations* at the Technical University of Munich [16]. The employed algorithm was optimized to reduce the necessary communication and a special physical layout of the LAN was used. The 1992 winners of the *Gordon Bell prize* [23] made experiments with up to 192 workstations using PVM. In 1993 the first prize of the *Mannheim SuParCup* was awarded to the author of this paper and to his collaborator Volker Strumpen for the genetic application described above, using a previous version of LOLA.

Encouraged by these scientific activities and awards we want to setup stable test environments to demonstrate that wide area massive parallelism works for *production purposes*. Furthermore we will investigate other kinds of *applications*.

However, one must always keep in mind, that at the present, and most likely also the next generation of WAN technology, there is a distinct bottleneck: WAN parallelism is only possible for a rather small albeit important class of applications. Unfortunately we must live with the realities of installed technology rather than utilize the most advanced techniques which are available. The ever shrinking budgets, however, are likely to promote cluster technologies.

Acknowledgements

I want to thank professors Lutz Richter and Kurt Bauknecht for an environment which is fun to work in and for their interest in my research. The financial support of Firma Siemens AG, München and Schweizer Bundesamt für Konjunktur und Wirtschaftsförderung made this work possible. Volker Strumpen helped with the implementation and installation of the molecular biology example and with the cited measurements. Edgar Lederer provided numerous ideas and suggestions. The Suparcup organized by professors Werner Meuer and Wolfgang Gentzsch gave the stimulus and a reward for this work. The measurements could be made when many friends and collaborators provided us with accounts on their workstations. All their help was and is of high value for our work.

References

[1] S. Ahuja, N. Carriero, D. Gelernter, Linda and Friends. *IEEE Computer* **19**(8), 1986, 26-34.

[2] M. Asawa and S. Misra, Robinhood: Resource Sharing by Time Stealing Between DOS PCs on a LAN, Technical report 91/2, James Cook University North Queensland, october 1991.

[3] J. Baud et al., SHIFT: The Scalable Heterogeneous Integrated Facility for HEP Computing. In Y. Watase (ed.), *Computing in High Energy Physics 91*, Universal Academy Press, Tokyo.

[4] R. D. Bjornson, Linda on Distributed Memory Multiprocessors. PhD Thesis and technical report YALEU/DCS/RR-931, University of Yale, november 1992.

[5] R. Bjornson, N. Carriero, D. Gelernter, T. Mattson, D. Kaminsky and A. Sherman. Experience with Linda. Technical report 866, department of computer science, Yale University, august 1991.

[6] R. Butler, E. Lusk, User's Guide to the p4 Parallel Programming System. Technical report, Argonne National Laboratory, august 1992.

[7] J. Corbin, The Art of Distributed Applications. Sun Technical Reference Library, Springer Verlag, 1991.

[8] C. H. Cap, Massively Parallel Computing in Wide Area Networks. Technical report, department of computer science, University of Zurich, 1993.

[9] C. H. Cap, Architekturelle Entscheidungen zur Unterstützung parallelen Rechnens in Workstation Clustern und deren Realisierung in existenten Produkten. Proceedings 2. ITG/GI Workshop Workstations, Hagen 1993, VDE Verlag.

[10] C. H. Cap and V. Strumpen, Efficient Parallel Computing in Distributed Workstation Environments. *Parallel Computing*, **19**(11), 1993.

[11] C. H. Cap and V. Strumpen, The Parform – A High Performance Parallel Platform for Efficient Computation in a Distributed Workstation Environment. Technical report IFI-TR 92.07 Technical report, department of computer science, University of Zurich, 1992.

[12] C. H. Cap and V. Strumpen, Massively Parallel Computing in the Internet – Entry to the SuParCup'93. Department of computer science, University of Zurich, 1993. This entry won the first prize of the SuParCup93.

[13] J. J. Dongarra, R. Hempel, A. J. G. Hey and D. W. Walker, A Proposal for a User-Level, Message-Passing Interface in a Distributed Memory Environment. Technical report CS-93-186, University of Tennessee, Knoxville, january 1993.

[14] G. Geist, M. Heath, B. Peyton and P. Worley. PICL a Portable Instrumented Communication Library. Technical report ORNL-TM-11130, Oak Ridge National Laboratory, Juli 1990. netlib@ornl.gov.

[15] G. Geist and V. Sunderam. The PVM system: Supercomputer Level Concurrent Computation on a Heterogeneous Network of Workstations. *Proceedings of the Sixth Distributed Memory Computing Conference*, IEEE Press, 1991,258–261.

[16] M. Griebel, W. Huber, T. Störtkuhl and C. Zenger, On the Parallel Solution of 3D PDEs on a Network of Workstations and on Vector Computers.

[17] R. Hempel, The ANL/GMD Macros (PARMACS) in FORTRAN for Portable Parallel Programming Using the Message Passing Programming Model, User's Guide and Reference Manual. Gesellschaft für Mathematik und Datenverarbeitung mbH (GMD), november 1991.

[18] HPCwire, Comparing the Efficiency of molecular biology sequence-analysis computations. Electronic Newsletter HPCwire, topic 815, March 1, 1993. Further source: SDSC's Computational Science Advances.

[19] R. J. Lipton, T. G. Marr and J. D. Welsh, Computational Approaches to Discovering Semantics in Molecular Biology. *Proc. IEEE* **77**(7) 1989, pp. 1056–1060.

[20] M. Litzkow, M. Livny and M. Mutka, Condor – Hunter of Idle Workstations. Technical report 730, University of Wisconsin, december 1987.

[21] H. P. Lüthi, J. Almlöf, Network Supercomputing: A distributed-concurrent direct SCF scheme. *Theoretica Chimica Acta* **84**, 1993, pp. 443–455.

[22] C. D. Marsan, NSF pursues computing without walls. *Federal Computer Week* **6**(35), 53, november 30, 1992.

[23] H. Nakanishi, V. Rego and V. Sunderam, Superconcurrent Simulation of Polymer Chains on Heterogeneous Networks. *Proceedings Supercomputer 92, Minneapolis*, IEEE Press, 1992, 561-569.

[24] Parasoft Corporation, Express Version 1.0: A Communication Environment for Parallel Computers. 1988.

[25] L. R. Revor, DQS users guide. Computing and Telecommunications Division, Argonne National Laboratory, september 1992.

[26] R. Roth, T. Setz, LiPS: A System for Distributed Processing on Workstations. Technical report, FB-14 Informatik, Universität des Saarlandes, february 1993.

[27] W. Scott, P. Arbenz, S. Vogel and H. P. Lüthi, Network Supercomputing. *EPFL Supercomputing Review*, november 1991.

[28] G. Shoinas, Issues on the Implementation of Programming Systems for Distributed Applications. Technical report, University of Crete, 1991.

[29] T. F. Smith, M. S. Waterman, Identification of Common Molecular Subsequences. *Journal of Molecular Biology* **147**, 1981, pp. 195–197

[30] C. Sprenger, Netzwerk-Computing. Diploma thesis, Institut für wissenschaftliches Rechnen, ETH Zurich, march 1992.

[31] V. Strumpen, Parallel Molecular Sequence Analysis on Workstations in the Internet. Technical report, department of computer science, University of Zurich, 1993.

[32] L. H. Turcotte, A Survey of Software Environments for Exploiting Networked Computing Resources. Technical report of the engineering research center for computational field simulation, june 1993.

[33] R. A. Wagner and M. J. Fischer, The String to String Correction Problem. *JACM* **21**(2) 1974, pp. 168–173.