Optimizing Parallel Applications for Wide-Area Clusters *

Henri E. Bal Aske Plaat Mirjam G. Bakker Peter Dozy Rutger F.H. Hofman
Technical Report IR-430, September 1997
Vrije Universiteit
Dept. of Mathematics and Computer Science
Amsterdam, The Netherlands
http://www.cs.vu.nl/albatross/

Abstract

Recent developments in networking technology cause a growing interest in connecting local-area clusters of workstations over wide-area links, creating multilevel clusters. Often, latency and bandwidth differences between local-area and wide-area network links are two orders of magnitude or more. With such a large difference, one would expect only very coarse grain applications to achieve good performance. Hence, most meta computing endeavors are targeted at job-level parallelism. To test this intuition, we have analyzed the behavior of several existing medium-grain applications on a wide-area multicluster. We find that, if the programs are optimized to take the multilevel network structure into account, most obtain high performance. The optimizations we used reduce intercluster traffic and hide intercluster latency, and substantially improve performance on wide-area multiclusters. As a result, the range of applications suited for a meta computer is larger than previously assumed.

Keywords: Cluster Computing, Meta Computing, Wide-area Networks, Communication Patterns, Performance Analysis, Parallel Algorithms

1 Introduction

One of the visions in the field of parallel computing is to exploit idle workstations to solve compute-intensive problems, such as those found in physics, chemistry, and biology. Today, many individual computers are connected by a local-area network (LAN) into a cluster. Advances in wide-area network technology make it possible to extend this idea to geographically distributed computers. By interconnecting multiple local clusters through a high-speed wide-area network (WAN), very large parallel systems can be built, at low additional cost to the user, creating a large parallel virtual machine. Several so-called meta computing projects (e.g., Legion [19], Condor [15], Dome [2]) try to create infrastructures to support this kind of computing. These projects investigate how to solve the problems that result from integrating distributed resources, such as heterogeneity, fault-tolerance, security, accounting, and load sharing.

The usefulness of a meta computing infrastructure depends on the applications that one can run successfully on them. Since wide-area links are orders of magnitude slower than local-area links, it is reasonable to

*This research is supported in part by a PIONIER grant from the Netherlands Organization for Scientific Research (N.W.O.).

expect that only applications that hardly communicate at all (i.e., embarrassingly parallel applications) will benefit from multiple WAN-connected clusters. Testing this hypothesis is the basis of our work. The research question we address here is *how parallel applications perform on a multilevel network structure*, in particular, on systems built out of both LANs and WANs. Existing meta computing projects often use applications with very coarse-grained (job-level) parallelism, which will perform well on any parallel system [19]. We investigate applications with a finer granularity, which were designed originally to run on a local cluster of workstations. In addition, we study optimizations that can be used to improve the performance on multilevel clusters.

The paper presents the following contributions:

- We present performance measurements for eight parallel programs on a wide-area multilevel cluster, and we identify performance problems. To the best of our knowledge, this is the first wide-area cluster benchmark consisting of applications with a nontrivial communication structure.
- For several applications we describe optimization techniques that substantially improve performance on a wide area system. The techniques appear general enough to be applicable to other programs as well.
- We conclude that, with the optimizations in place, many programs obtain good performance, showing that it is beneficial to run parallel programs on multiple WAN-connected clusters. This conclusion is surprising, given that our system's WAN is almost two orders of magnitude slower than its LAN.
- Since adequate performance can be obtained for a variety of nontrivial applications, our work indicates that meta computing efforts like Legion and Dome are all the more worth while.

The testbed we use consists of four cluster computers located at different universities in the Netherlands, connected by a wide area ATM network (see Figure 17). The four clusters use identical processors (a total of 136 Pentium Pros) and local networks (Myrinet [9]). The advantage of this homogeneous setup is that we can study the impact of LAN and WAN speed on application performance without having to be concerned about other factors (e.g., differences in CPU types or speeds). We isolate one important performance factor and study its impact. The experimental testbed is designed specifically to allow this kind of research. In contrast, most meta computing projects use existing workstations, which leads to more heterogeneous testbeds.

Multilevel clusters are somewhat similar to NUMA (Non Uniform Memory Access) machines, in that the communication latency is non uniform. However, the relative difference between between sending a message over a LAN or a WAN is much higher than that between accessing local or remote memory in a NUMA (for example, in an SGI Origin2000 it is typically only a factor of 2–3 [25]).

The outline of the rest of the paper is as follows. In Section 2 we describe our experimental setup in more detail. In Section 3 we present the suite of applications. In Section 4 we give the performance results for the original and optimized programs. In Section 5 we discuss these results and present the general lessons we have learned. In Section 6 we look at related work. Finally, in Section 7 we present our conclusions.

2 Experimental setup

In this section we first describe the experimental system used for our research, including the hardware and systems software. Also, we give basic performance numbers for the system. (Additional information on the system can be found at *http://www.cs.vu.nl/~bal/das.html*).

The single most distinguishing feature of a wide-area multicluster is the large difference in latency and bandwidth of the communication links. Modern LANs have an application-to-application latency of 10–100 microseconds [29, 36], whereas a WAN has a latency of several *milli*seconds. The bandwidth of a high-speed LAN is about 10–100 Mbyte/sec; for WANs bandwidth varies greatly. Currently, an average sustained bandwidth of 100–5000 kbyte/sec may be a reasonable assumption for most academic environments. (Note that these are application-level figures, not hardware-level.) It is expected that the relative differences between LAN and WAN performance will persist, if only for latencies, because the speed of light is becoming a limiting factor.

Our wide-area system consists of four clusters, of which two were operational when the measurements were performed. To be able to obtain results for four clusters and to be able to vary bandwidth and latency, a single 64-node cluster was split into four smaller clusters, and configured to yield similar performance behavior as the real wide-area system. This experimentation system has been validated by comparing its performance for two clusters with the real wide-area system. We first describe the real wide-area system and then the difference with the experimentation system.

The system has been designed by the Advanced School for Computing and Imaging (ASCI).[1] It is called DAS, for Distributed ASCI Supercomputer. The main goal of the DAS project is to support research on wide-area parallel and distributed computing. The structure of DAS is shown in Figure 17. Each of the four participating universities has a local cluster of 200 MHz Pentium Pros. Three sites have a 24 node cluster; the VU Amsterdam has a 64 node cluster. Each node contains 64 MByte of memory, 256 KByte L2 cache, and a 2.5 GByte local disk. The system has 144 computers in total (136 compute nodes, four file servers, and four gateway servers). It has 10 GByte DRAM, 376 GByte disk, and 28.8 GFLOPS peak performance. The operating system used on DAS is BSD/OS from BSDI. The nodes of each cluster are connected by Myrinet and Fast Ethernet. Myrinet is used as fast local interconnect, using highly efficient protocols that run entirely in user space. Fast Ethernet is used for operating system traffic (including NFS, remote shell, and so on). The four clusters are connected by a wide-area ATM network. The network uses a Permanent Virtual Circuit between every pair of sites. At present, they have a Constant Bit Rate bandwidth of 6 Mbit/sec. Each cluster has one gateway machine, containing a ForeRunner PCA-200E ATM board. All messages for machines in a remote cluster are first sent over Fast Ethernet to the local gateway, which forwards them to the remote gateway, using IP.

Currently, two of the four clusters (at Delft and the VU Amsterdam) are operational and connected by the wide-area ATM link. We have measured that the roundtrip application-level latency over this ATM link is 2.7 milliseconds; application-level bandwidth was measured to be 4.53 Mbit/sec. For reference, the same benchmark over the ordinary Internet on a quiet Sunday morning showed a latency of 8 milliseconds and a

[1] The ASCI research school is unrelated to, and came into existence before, the Accelerated Strategic Computing Initiative.

bandwidth of 1.8 Mbit/sec.

To be able to run four cluster experiments the 64 node cluster at the VU in Amsterdam has been split into four smaller sub-clusters. The four sub-clusters are connected by a local ATM network. Each sub-cluster has one machine that acts as gateway; as in the real wide-area system, the gateway is dedicated and does not run application processes. With four sub-clusters, each sub-cluster thus consists of at most 15 compute nodes and one gateway. Each gateway contains the same ATM interface board as in the real system. The ATM firmware allowed us to limit bandwidth by transmitting extra idle cells for each data cell. This was used to configure the boards to deliver the same bandwidth as measured for the real system. In addition, we have increased the latency for communication over the local ATM links by modifying the low-level communication software running on the gateway. When the gateway receives a message over the ATM link, it spins for an extra 600 microseconds before it sends the message to the destination machine, creating a total round trip latency of 2.7 milliseconds. (Since the gateway does not run user processes, spinning does not waste CPU cycles for the application.)

The only important difference between the real system and the experimentation system is the wide area ATM link. Except for this ATM link, the two systems are the same; the same executable binaries are used on both systems. The experimentation system has been validated by running all applications on the two-cluster experimentation system and on the wide-area system consisting of the clusters in Delft and VU Amsterdam, using 16 compute nodes per cluster. The average difference in run times is 1.14% (with a standard deviation of 3.62%), showing that the wide-area ATM link can be modeled quite accurately in the way we described above.

The applications used for the performance study are written in Orca, a portable, object-based parallel language, in which processes communicate through shared objects. One of the more interesting features of the Orca system is how it exploits broadcasting. To achieve good performance, the Orca system replicates objects that have a high read/write ratio. Invocations on non-replicated objects are implemented using Remote Procedure Calls (RPCs). For replicated objects, read-only operations are executed locally. Write-operations on replicated objects are implemented using a write-update protocol with function shipping: the operation and its parameters are broadcast, and each machine applies the operation to its local copy. To keep replicated objects consistent, a totally-ordered broadcast is used, which guarantees that all messages arrive in the same order on all machines. Broadcasting is implemented using a single sequencer machine to order messages. We will explain the communication behavior of the Orca applications mainly in terms of RPCs and broadcasts, although at the programming level this difference is hidden: the Orca programmer just uses one abstraction (shared objects).

The Orca Runtime System (RTS) on DAS uses Myrinet for *intracluster* communication (communication between processors in the same cluster). The low-level software is based on Illinois Fast Messages [29], which we extended to support fast broadcasting, amongst others [4]. A separate study showed that communication in a local Myrinet cluster is sufficiently fast to run several communication-intensive applications efficiently [24].

For *intercluster* (wide-area) communication, the Orca system uses the ATM network. RPCs are implemented by first sending the request message over Fast Ethernet to the local gateway machine (which is not part of the Myrinet network). The gateway routes the request over the ATM link to the remote gateway, using

Benchmark	latency		bandwidth	
	Myrinet (LAN)	ATM (WAN)	Myrinet (LAN)	ATM (WAN)
RPC (non-replicated)	40 µs	2.7 ms	208 Mbit/s	4.53 Mbit/s
Broadcast (replicated)	65 µs	3.0 ms	248 Mbit/s	4.53 Mbit/s

Table 1: Application-to-application performance for the low-level Orca primitives.

TCP. The remote gateway delivers the message to the destination machine. Reply messages are handled in a similar way. Broadcasts pose some problems. Implementing totally-ordered broadcasting efficiently on a WAN is challenging. The centralized sequencer works well for the local cluster, but becomes a major performance problem on a WAN. Our current solution is to use a distributed sequencer (one per cluster) and allow each cluster to broadcast in turn. On a WAN, this approach is more efficient than a centralized sequencer, but for several applications the latency is still too high. For certain applications a specialized mechanism can be used (see Section 4), but it remains to be seen if more efficient general protocols can be designed.

Table 1 gives Orca's low-level performance figures, for intracluster and intercluster communication, for non-replicated objects and replicated objects. Latency is measured using null operations, bandwidth using a sequence of 100 KByte messages. The remote object invocation benchmark measures the latency to invoke an operation on a remote object. The replicated-object invocation benchmark measures the latency to update an object that is replicated on 60 machines, which involves broadcasting the operation to all these machines. The performance gap between the LAN and WAN is large, so much so that even a low communication volume is expected to cause applications to experience serious slowdowns.

3 Applications

We have selected eight existing parallel Orca applications for our performance study. The applications were originally developed and tuned to run on a single Massively Parallel Processor or local cluster; the applications were designed for an architecture with a single-level communication network.

The goal of this study is to see whether medium grain communication can (be made to) work on a wide-area multilevel cluster—not to achieve the best absolute speedup for a particular system or application. Therefore, applications and problem sizes were chosen to have medium grain communication: not trivially parallel, nor too challenging. For our set of input problems the applications obtain an efficiency between 40.5 and 98 percent when run on the local 64-node Myrinet cluster. The programs represent a wide variety of application domains and include numerical, discrete optimization, and symbolic applications.The total number of lines of Orca code is 11,549. Table 2 lists the eight applications, together with a brief characterization, describing their type, their main communication pattern, and some basic performance data. Most applications primarily use point-to-point communication, except ACP and ASP, which use broadcast. The performance data were collected by running the programs on one Myrinet cluster with 64 nodes, using the input problems described later. We give both the total number of RPCs or broadcasts per second (issued by all processors together) and the total amount of user data sent per second. Finally, we give the speedup on 64 nodes. Essentially, the table shows that on a single Myrinet cluster all algorithms run reasonably efficient.

program	type	communication	# RPC/s	kbytes/s	# bcast/s	kbytes/s	speedup
Water	*n*-body	exchange	9,061	18,958	48	1	56.5
TSP	search	work queue	5,692	285	134	11	62.9
ASP	data-parallel	broadcast	3	49	125	721	59.3
ATPG	data-parallel	accumulator	4,508	18	64	0	50.3
IDA*	search	work stealing	8,156	202	477	1	62.1
RA	data-parallel	irregular	240,297	8,493	296	0	25.9
ACP	iterative	broadcast	77	826	1,649	557	37.0
SOR	data-parallel	neighbor	18,811	67,540	326	2	46.3

Table 2: Application characteristics on 64 processors on one local cluster.

An important issue is which input sizes to use for each application. For applications where the amount of computation grows faster with problem size than communication, choosing a bigger problem size can reduce the relative impact of overheads such as communication latencies. (Basically, we are applying Amdahl's law here, improving speedup by reducing critical path [8, 7].) In our situation, we could have increased the problem sizes to compensate for the slowness of the WANs. However, we have decided not to do so, since determining the impact of the WAN is precisely what we want to do. Thus, we believe and expect that the speedup figures that follow can be improved upon.

In addition to the size of the input problem, the number of processors and clusters also has an effect on efficiency (which is speedup divided by the number of processors). Increasing the number of processors generally increases the number of peers with which each processor communicates. Furthermore, each processor has a smaller share of the total work. Both factors reduce efficiency.

4 Application performance on DAS

In this section we will discuss the performance of the eight applications on the wide-area system. For each application we give the speedup (relative to the one-processor case) of the original program, running on 1, 2, and 4 clusters, using an equal number of processors per cluster. We use 1, 8, 16, 32, and 60 computational nodes. (We cannot use 64 computational nodes, since 4 gateway machines are needed for the 4-cluster runs.) Speedups are measured for the core parallel algorithms, excluding program startup time. In addition, we describe how we optimized the programs for the wide-area system and we give the speedup of the optimized program (relative to itself). For brevity we do not dwell on individual application characteristics here. More details can be found in a technical report available from *http://www.cs.vu.nl/albatross/* [5]. Section 5 further summarizes and discusses the optimizations.

4.1 Water

The Water program is based on the "n-squared" Water application from the Splash benchmark suite [33]. It is a classical *n*-body simulation program of the behavior of water molecules in an imaginary box. Each processor is assigned an equal number of water molecules.

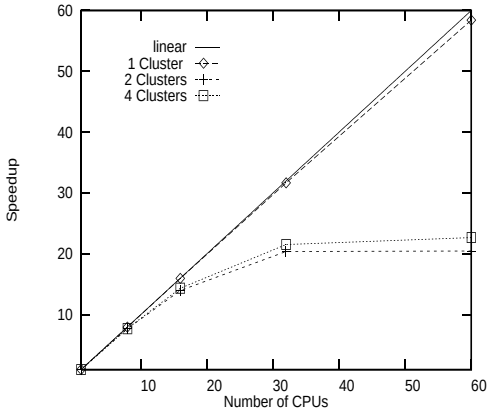


Figure 1: Speedup of Water

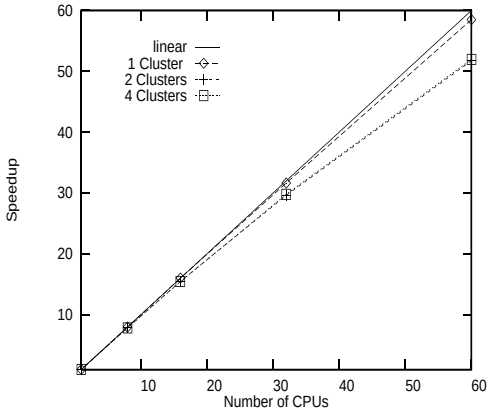


Figure 2: Speedup of optimized Water

The running time of the algorithm is quadratic in the number of molecules, and also depends on the number of iterations (which determines the desired precision). The total communication volume increases with the number of iterations, the number of processors and the number of molecules, though linear, not quadratic. Therefore, grain size increases with problem size, as with most other applications.

The performance of the original program is shown in Figure 1, using an input problem with 4096 molecules and computing two time steps. The *x*-axis of the figure shows the total number of CPUs being used, so, for example, the right-most data point on the 4-cluster line uses four clusters of 15 CPUs each. The Water program suffers from a severe performance degradation when run on multiple clusters. The performance problem is the exchange of molecule data. At the beginning of each iteration, every processor gets the positions of the molecules on the next $p/2$ processors (where p is the number of processors). When all data have arrived, the processor waits until the other $p/2$ processors have received the data from this processor. Likewise, at the end of the iteration, all forces computed are sent to the next $p/2$ processors and summed there. Since a substantial part of these messages cross cluster boundaries, the performance of the program degrades on multiple clusters.

The optimization we applied essentially does caching of molecule data at the cluster level. With the original program, the data for a given molecule are transferred many times over the same WAN link, since multiple processors in a cluster need the data. The optimization avoids sending the same data over the same WAN link more than once. For every processor *P* in a remote cluster, we designate one of the processors in the local cluster as the *local coordinator* for *P*. If a process needs the molecule data of processor *P*, it does an intracluster RPC to the local coordinator of *P*. The coordinator gets the data over the WAN, forwards it to the requester, and also caches it locally. If other processors in the cluster ask for the same data, they are sent the cached copy. A similar optimization is used at the end of the iteration. All updates are first sent to the local coordinator, which does a reduction operation (addition) on the data and transfers only the result over the WAN. (Coherency problems are easily avoided, since the local coordinator knows in advance which

processors are going to read and write the data.)

The speedups for the optimized program are shown in Figure 2. The new program achieves a speedup on four 15-node clusters that is close to the single 60-node cluster speedup (which is the best obtainable performance). The optimization shows that substantial performance improvements are possible when the multilevel network structure is taken into account.

4.2 The Traveling Salesman Problem

The Traveling Salesman Problem (TSP) program computes the shortest path starting from one city and visiting all other cities in a given set exactly once, using a branch-and-bound algorithm. The program uses master/worker parallelism, and is used here to study the behavior of a simple dynamic load balancing scheme with a centralized job queue. A master process generates jobs for a number of worker processes (one per processor). The jobs are placed in a FIFO queue, stored in a shared object located on the manager's machine. A job contains an initial path of a fixed number of cities. The worker processes take jobs from the queue and compute the lengths of all possible paths starting with the initial path. The program keeps track of the length of the current best solution found so far. Partial routes that are longer than this "global minimum" are pruned. The value of the global minimum is stored in a shared object that is replicated on all machines (because it is read frequently). In this experiment, to prevent non-deterministic results, the global bound was fixed in this experiment.

The time complexity and grain size of the algorithm increase exponentially with job size (number of cities to solve for). The total communication volume is not influenced by the number of processors or clusters. The communication volume that crosses cluster boundaries is linearly related to the number of clusters. (Since computation increases faster—exponentially—than communication with growing problem size, grain increases as well. In this parallelization, grain size can be controlled by adjusting the depth to which the master generates jobs. Too coarse a grain causes load imbalance.) Table 2 shows that the communication volume is quite low for our input size.

The speedup for the TSP program is shown in Figure 3, using a 17-city problem. The performance of the program on multiple clusters is mediocre. The overhead of intercluster communication is caused primarily by the work distribution and the global minimum. The program uses a shared job queue object that is stored on one processor, so processors on remote clusters need to do an intercluster RPC each time they need a new job. With four clusters, about 75% of the jobs will be sent over the WAN.

For optimization, we decreased the intercluster communication by using an alternative work distribution scheme. Instead of dynamic work distribution through a centralized job queue we used a static distribution over the clusters, implemented with a local job queue per cluster. In this way, intercluster communication is reduced substantially, but load imbalance is increased. Nevertheless, this modification improved performance substantially, as can be seen in Figure 4. The graph shows a small amount of super-linear speedup for the one-cluster case. To avoid non-determinacy and the possibility of algorithmic super-linear speedup, the global bound was fixed in advance in these experiments. We therefore attribute the super-linear speedup to the increase in the amount of fast cache memory as more machines are used.

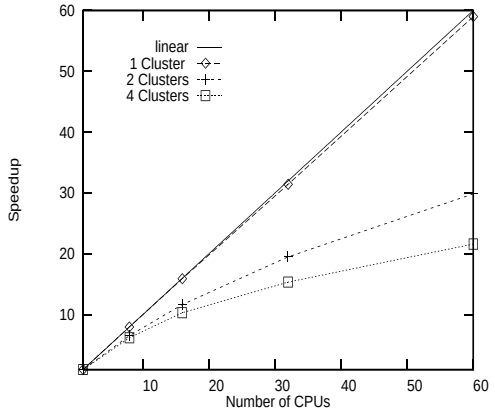


Figure 3: Speedup of TSP

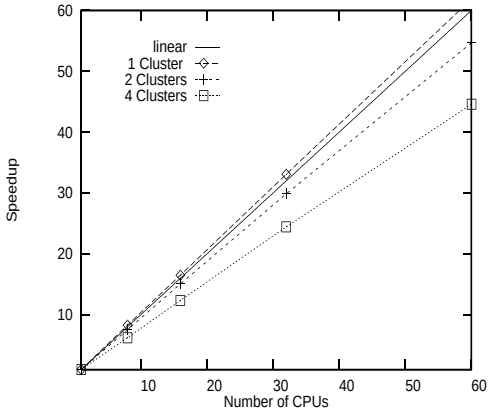


Figure 4: Speedup of optimized TSP

4.3 All-pairs Shortest Paths Problem

The goal of the All-pairs Shortest Paths (ASP) problem is to find the shortest path between any pair of nodes in a graph. The program uses a distance matrix that is divided row-wise among the available processors.

The running time of the algorithm is cubic in n, communication is quadratic in n. Total communication volume increases with the number of processors.

The original program (using an input problem with 3,000 nodes) obtains a poor performance on multiple clusters (Figure 5). This can be explained by the communication behavior. The program performs 3,000 iterations, and at the beginning of each iteration one of the processors broadcasts a row of the matrix to all other processors. The other processors cannot start the iteration until they have received this row, and the sender has to wait for a sequence number before continuing. On a local cluster, broadcasting is efficient (see Table 1). On the wide-area system, however, every broadcast message has to cross the WAN links. Even worse, due to the total ordering of broadcasts in the Orca system (see Section 2), broadcast messages require additional sequencer messages that usually go over the WAN.

The broadcast mechanism can be optimized by noting that all broadcasts are sent in phases: first processor 1 computes its rows and broadcasts them, then processor 2, etc. We can take advantage of this pattern by implementing a different sequencer mechanism: we can create a centralized sequencer and migrate it to the cluster that does the sending, so that the sender receives its sequence number quickly, and can continue. This optimization works well, allowing pipelining of computation and communication.Other optimizations are to use a dedicated node as cluster sequencer, and to use a spanning tree to forward requests, reducing sequentialization at the cluster communicator. The performance for the optimized program is given in Figure 6.

Another problem with this application is that the total communication volume of the program is relatively large, and grows with the number of processors, causing the wide-area link to become saturated. For the runs shown in the graphs this effect is not present; it occurs only for smaller problems, more machines, or with a

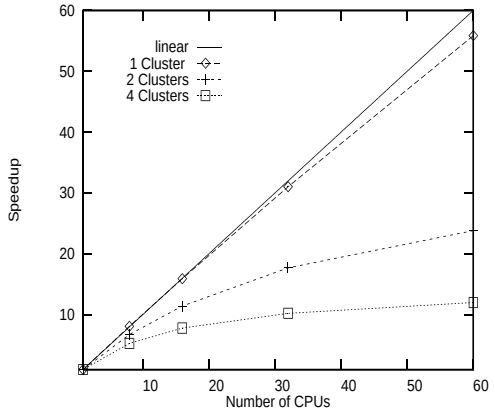


Figure 5: Speedup of ASP

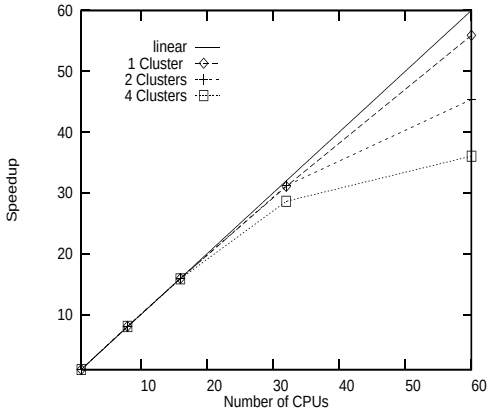


Figure 6: Speedup of optimized ASP

lower inter-cluster bandwidth.

4.4 Automatic Test Pattern Generation

Automatic Test Pattern Generation (ATPG) is a problem from electrical engineering [22]. The goal is to compute a set of test patterns for a combinatorial circuit, such that the patterns together test all (or most) gates in a circuit. The program parallelizes ATPG by statically partitioning the gates among the processors. Due to the static partitioning of the work, the program does little communication. The processors communicate to keep track of how many test patterns have been generated and how many gates the tests cover.

The speedup for the ATPG program is shown in Figure 7. The program obtains a high efficiency on one cluster (even on 60 CPUs). On multiple clusters, the efficiency decreases only slightly, because the program does little communication. On slower networks (e.g., 10 ms latency, 2 Mbit/s bandwidth) the performance of the ATPG program is significantly worse (not shown), and the following straightforward optimization has been implemented. In ATPG, each processor updates a shared object (containing the number of test patterns) every time it generates a new pattern, resulting in many RPCs. On multiple clusters, many of these RPCs go over the wide-area network and therefore cause a performance degradation. The solution to this problem is straightforward. The number of test patterns and gates covered is only needed (and printed) at the end of the program execution. It is therefore possible to let each processor accumulate these numbers locally, and send the totals in one RPC when it is finished with its part of the work. A further optimization is to let all processors of one cluster first compute the sum of their totals, and then send this value to the first processor using a single RPC. In this way, intercluster communication is reduced to a single RPC per cluster. For the bandwidth and latency settings of this experiment, the speedups were not significantly improved, however (see Figure 8).

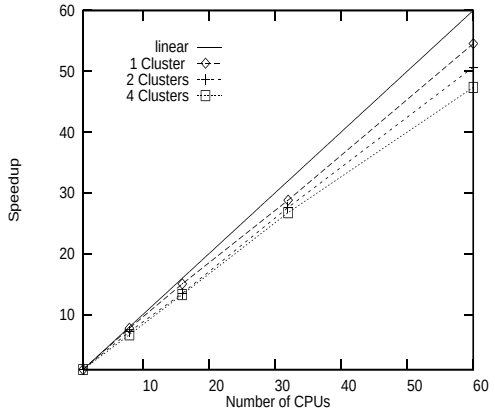


Figure 7: Speedup of ATPG

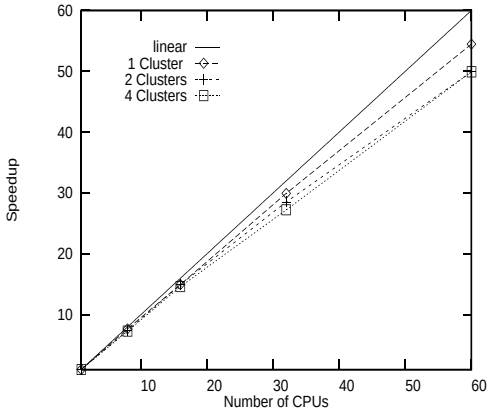


Figure 8: Speedup of optimized ATPG

4.5 Retrograde Analysis

Retrograde Analysis (RA) is a technique to enumerate end-game databases, of importance in game-playing. RA is based on backwards reasoning and bottom-up search. It starts with end positions, whose game-theoretical values are known (e.g., checkmate in chess). From these positions, RA works its way back to compute the values of other positions. The problem can be parallelized by dividing the positions among the processors. The resulting program sends many small messages, which can be asynchronous (the sender does not wait for a reply). This allows them to be combined into fewer, larger, messages. This *message combining* optimization greatly improves performance [3]. The program computes a 12-stone end-game database for Awari, an African board game. The performance of the original parallel RA program is shown in Figure 9.

The running time of the algorithm is exponential in the number of pieces of the database; regrettably, communication scales exponentially too. The total communication volume also increases with the number of processors though linear, not exponentially. Grain size is small, and message combining is needed to increase it to acceptable levels.

The performance of the program drops dramatically when it is run on multiple clusters. The speedup on four 15-node clusters is even less than 1. The reason is the large amount of communication. Unfortunately, the communication pattern is highly irregular: every processor sends messages (containing updates for the database) to all other processors in a highly unpredictable way. It is therefore difficult to reduce the inter-cluster communication volume. We optimized the program by applying message combining at the cluster level. If a processor wants to send a message to a machine in a remote cluster, it first sends the message to a designated machine in its own cluster. This machine accumulates all outgoing messages, and occasionally sends all messages with the same destination cluster in one large intercluster message. The performance of the program improves substantially by this optimization, especially for databases with fine grain communication (the grain increases for larger databases, and for very large databases the extra cluster combining overhead even defeats the gains). The execution time on four 15-node clusters improved by a factor of 2.

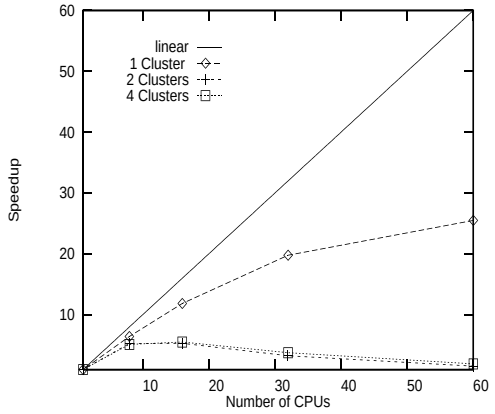


Figure 9: Speedup of RA

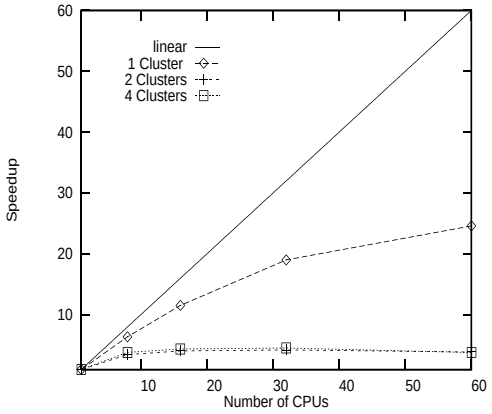


Figure 10: Speedup of optimized RA

Still, the optimized program is slower on multiple clusters than on one (15-node) cluster, making it unsuitable for the wide-area system, at least with our bandwidth and latency parameters.

4.6 Iterative Deepening A*

Like TSP's branch-and-bound, Iterative Deepening A* [23] (IDA*) is a combinatorial search algorithm. IDA* is used for solving random instances of the 15-puzzle. The program is useful to study a more advanced load balancing mechanism: a distributed job queue with work stealing. IDA* repeatedly performs a depth-first search. Initially, the maximum search depth is set to a lower bound on the number of moves needed to solve the puzzle. As long as no solution is found, the maximum depth is increased, and the search continues with the next iteration. Pruning is used to avoid searching useless branches.

IDA* is parallelized by searching different parts of the search tree concurrently. Non-determinism is avoided by searching to find *all* solutions at a certain depth. Each process has a local job queue from which it gets its jobs. When there are no local jobs, the process tries to steal a job from another process. For each job, the worker either prunes this branch of the search tree or it expands the children by applying valid moves. The resulting board positions are placed into the local job queue. At the end of each iteration, the workers start running out of work, and load balancing occurs until the iteration has ended, after which a new search is started with a higher search bound. In a way the program performs a global synchronization at the end of each iteration, making sure no processor is working anymore.

The algorithm has a time complexity which increases exponentially with job size (the number of moves it takes to solve a puzzle instance). Total communication volume and grain size are determined by the amount of load imbalance and by how many other processors are trying to steal jobs. Thus, the communication volume that crosses cluster boundaries increases strongly with the number of clusters. However, as job size increases, computation increases even stronger, so the overall grain also grows with job size.

Figure 11 shows the speedups for the IDA* program. (The 2-cluster line overlaps mostly with the 4-

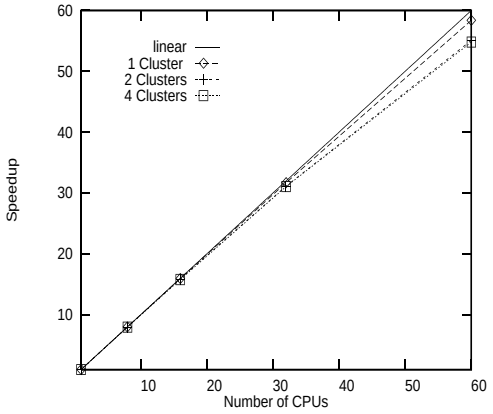


Figure 11: Speedup of IDA*

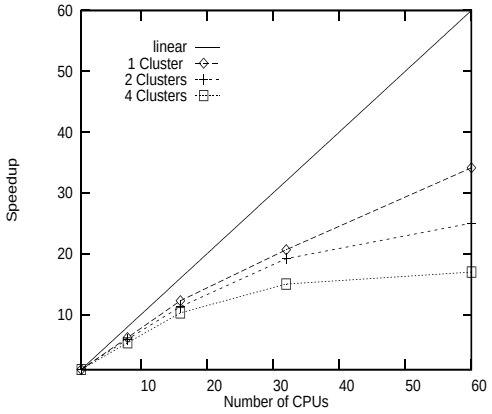


Figure 12: Speedup of ACP

cluster line.) The program performs quite well. Nevertheless, an optimization was introduced in the work stealing strategy. To steal a job, a worker process asks several other processors in turn for work, until it succeeds. The set of processors it asks for work is fixed and is computed by adding $1, 2, \ldots, 2^n$ modulo p with $2^n \leq p$ (with p the number of processors used) to the local process number. This strategy works poorly for the highest-numbered process in a cluster. Such processes always start looking for work in remote clusters first to steal jobs.

To solve the problem, we applied two optimizations. First, we changed the order of work-stealing. The optimized program first tries to steal jobs from machines in its own cluster. The second optimization tries to reduce the number of job-requests by maintaining load-balancing information. The IDA* program uses a simple termination-detection algorithm, which requires every worker process to broadcast a message when it runs out of work or when it becomes active again (because it received new work). It therefore is possible to let each process keep track of which other processes are idle, and to avoid sending job-requests to processors that are known to be idle.

We have measured that the maximal number of intercluster RPCs per processor has almost been halved. The speedup on multiple clusters has hardly changed, however (not shown). IDA* makes a limited number of work steal requests, since the load balance is relatively good. TSP showed that a centralized job queue does not work well on a LAN/WAN system. The optimization was to distribute the work queue over the clusters, with a static work division. For IDA*, the original distributed work queue worked well on our system due to a relatively good load balance. The optimization, attempting to steal within one's own cluster first, may still be of use for finer grain applications, applications with a larger load imbalance, or slower networks.

4.7 Arc Consistency Problem

Algorithms for the Arc Consistency Problem (ACP) can be used as a first step in solving Constraint Satisfaction Problems. The ACP program takes as input a set of n variables and a set of binary constraints defined on

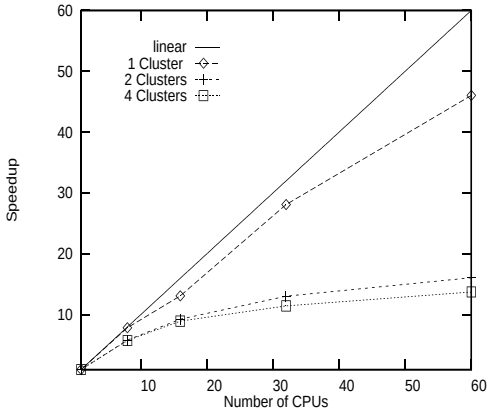


Figure 13: Speedup of SOR

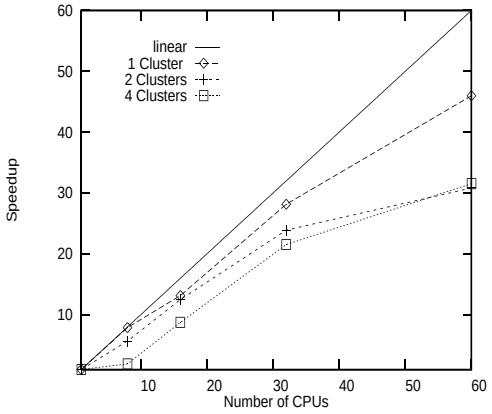


Figure 14: Speedup of optimized SOR

some pairs of variables, that restrict the values these variables can take. The program eliminates impossible values from the domains by repeatedly applying the constraints, until no further restrictions are possible. The program is parallelized by dividing the variables statically among all processors.

The performance for a problem with 1,500 variables is shown in Figure 12. If a processor has modified one of the variables assigned to it, it must inform all other processors. The program implements this behavior by storing the variables in a shared replicated object, which is updated using broadcasts. ACP performs many small broadcasts, causing much traffic for cluster gateways. Still, the speedup on two and four clusters exceeds that for a single cluster of the same number of processors. In ACP, the sender of a broadcast message need not wait until the message has arrived at all processors. Therefore, a possible optimization for ACP is to use asynchronous broadcasts. This idea has not been implemented, so no results are shown.

4.8 Successive Overrelaxation

Successive Overrelaxation (SOR) is an iterative method for solving discretized Laplace equations on a grid. It is used as an example of nearest neighbor parallelization methods. The parallel algorithm we use distributes the grid row-wise among the available processors. The speedups for the original SOR program (using a 3500 $\times$ 900 grid as input, and a termination precision of 0.0002, leading to 52 iterations) are shown in Figure 13.

The SOR program does a significant amount of communication. The time complexity of the algorithm is quadratic, and also depends on the number of iterations (determined by the desired precision). As with most other applications, grain size increases with problem size.

The SOR program logically organizes the processes in a linear array. At the beginning of every iteration, each processor (except the first and last one) exchanges a row with both its left and right neighbor. This overhead already slows down the program on a single cluster (where all communication goes over the fast Myrinet network). The performance on multiple clusters is much worse, because the first and last processor of each cluster have to send a row to a remote cluster. These processors block in an intercluster RPC at the

beginning of every iteration, causing delays in this synchronous algorithm.On four 16-node clusters, for example, this RPC costs 5 milliseconds, while a single iteration costs 100 milliseconds, so the relative overhead is noticable. Eventually, this slows down other processors in the cluster, because of data dependencies.

To improve the performance, two optimizations were found useful. First, we modified the program to overlap computation and communication. At the beginning of an iteration, rows are sent to the neighbors, and instead of waiting to receive those rows, the program first computes the inner rows (which are independent due to the red-black scheme). Once those have been completed, a receive for the outer rows is performed, after which they are computed. Orca does not allow this kind of split-phase send/receive to be expressed. Therefore, we rewrote the program in C, using the lower level send and receive primitives of the Orca RTS. This optimization works well for larger problem sizes, where there is enough computational work to overlap with the intercluster communication. The resulting performance is a modest improvement, not shown. A second optimization is described next.

In red/black SOR, the distributed matrix is kept consistent by exchanging all boundary rows after each iteration. In [12] a different relaxation scheme is presented, called *chaotic relaxation.* It is shown that even if processors exchange rows at random, convergence can still be guaranteed in most cases. Applying this idea to multicluster SOR is straightforward. At cluster boundaries, some row exchanges are skipped, reducing the communication overhead substantially, at the cost of slower convergence. Within a cluster all row exchanges proceed as usual. By limiting the number of row exchanges that are skipped, convergence speed can be preserved. In our experiment, using up to 4 clusters, we dropped 2 out of 3 intercluster row exchanges, which increased the number of iterations in the convergence process by 5–10%. As the number of clusters increases, so does the relative number of row exchanges that are dropped; changes propagate slower through the matrix, and convergence becomes slower. We found that for a modest number of clusters convergence speed can be preserved easily.

There is another opportunity for optimization in SOR. Orca is based on RPCs, causing communication to be synchronous. By using asynchronous messages, communication and computation can be pipelined. Using a re-implementation of SOR in C, we found that for some problem sizes the program could be sped up somewhat, although not by as much as the Orca version that reduces intercluster row exchanges. (This version is not shown in the graphs.)

Figure 14 shows that the trade-off of intercluster communication versus convergence speed works well. Multicluster speedup has been improved substantially, and with the optimization four 15-processor clusters are now faster than one.

5 Discussion

This section discusses the results of the experiments. First, the magnitude of the performance improvements is analyzed. Next, the different techniques that were used to achieve these improvements are characterized.

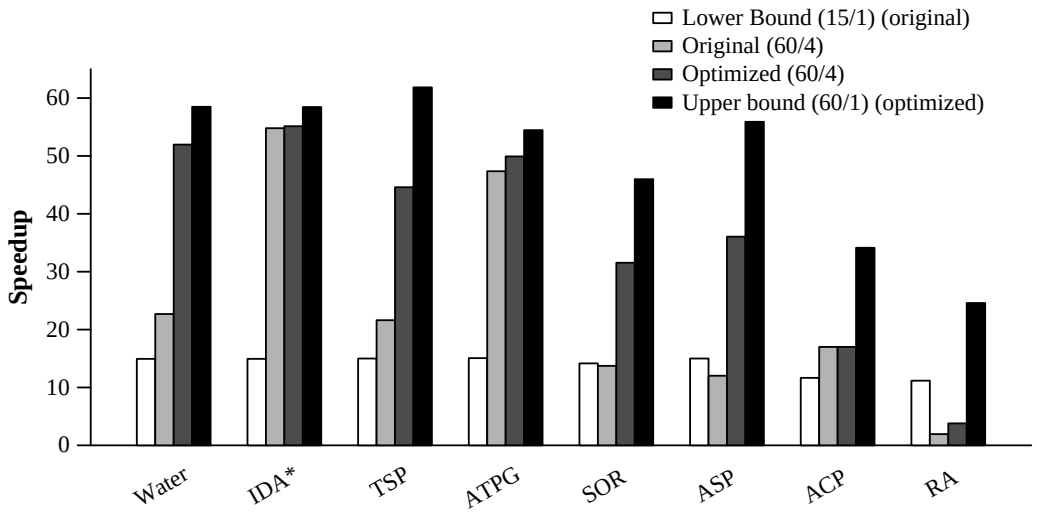


Figure 15: Four-Cluster Performance Improvements on 15 and 60 processors

5.1 Performance Assessment

To assess the efficiency of an application running on C clusters with P processors each, two metrics are useful. First, the best possible performance is that of the same program running on a single cluster with $C \cdot P$ processors. Second, for wide-area parallel programming to be useful, at the bare minimum one would want the program to run faster on C clusters with P machines each than on one cluster with P machines (i.e., using additional clusters located at remote sites should not hurt performance). These two metrics thus give an upper bound and a lower bound, or optimal versus acceptable performance.

Figure 15 shows the performance of the original and optimized programs for all eight applications, together with the lower and upper bound on acceptable performance. We use up to four clusters with 15 nodes each. For each application, the first bar is the speedup on a single 15-node cluster, which is the lower bound for acceptable speedup. The last bar of each application is the speedup on a single 60-node cluster, which is the upper bound on performance. The second and third bars are the speedups for the original and optimized programs on four 15-node clusters. (The first two bars are for the original program and the last two bars are for the optimized program.)

For completeness and comparison, Figure 16 shows the two-cluster runs using the wide-area system consisting of the clusters in Delft and the VU Amsterdam. On two clusters, performance is generally closer to the upper bound.

As can be seen, for the original programs five applications run faster on four clusters than on one cluster, although for only two applications (IDA* and ATPG) the performance approaches the upper bound; all others perform considerably worse. For four applications (Water, TSP, ASP, and SOR), the optimizations cause the performance to come close to the upper bound; the optimized versions experience a modest performance

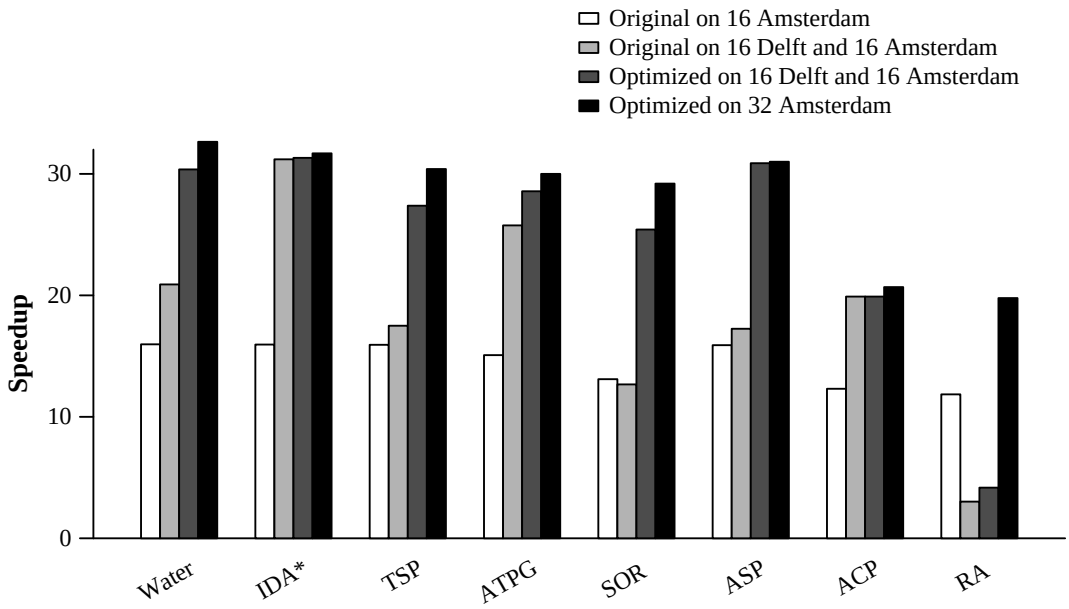


Figure 16: Two-Cluster Performance Improvements on 16 and 32 processors Delft/Amsterdam

degradation from the WAN links. Finally, by comparing the second and third bar for each application we can determine the impact of the performance optimizations. For five applications (Water, TSP, SOR, ASP, and RA) the impact is substantial, with an average speedup increase of 85 percent.

5.2 Optimizations and Communication Patterns

We will now discuss the communication patterns of the applications and the optimizations. They are summarized in Table 3 , and the reduction in intercluster traffic can be seen from tables 4 and 5.

Since the performance of the WAN links is much lower than that of the LAN links, one important class of optimizations is reduction of the inter-cluster communication volume. Alternatively, measures to make better use of the available bandwidth can be taken by hiding intercluster latency. The optimizations that were applied to the algorithms either reduce intercluster communication or try to mask its effect.

For five applications (Water, IDA*, TSP, ATPG, SOR), we were able to reduce intercluster communication. Both job queue optimizations fall into this category. The simplest scheme, a physically centralized work queue, leads to performance problems on a multilevel cluster. The optimization distributes the job queue over the clusters, dividing work statically over the cluster queues. This trades off static versus dynamic load balancing, substantially reducing intercluster communication. The resulting increase in load imbalance can be reduced by choosing a smaller grain of work, at the expense of increasing intracluster communication overhead (TSP). For the fully distributed work-stealing scheme—giving each *processor* its own queue—it is

Application	Communication structure	Improvements
Water	All to all exchange	Cluster cache
ATPG	All to one	Cluster-level reduction
TSP	Central job queue	Static distribution
IDA*	Distributed job queue with work stealing	Steal from local cluster first "Remember empty" heuristic
ACP	Irregular broadcast	None implemented
ASP	Regular broadcast	Sequencer migration
RA	Irregular message passing	Message combining per cluster
SOR	Nearest neighbor	Reduced ("chaotic") relaxation

Table 3: Patterns of Improvement

obviously more efficient to look for work in the local cluster before doing an expensive intercluster lookup, and also, to remember which queues were empty at a previous attempt (IDA*). For associative all-to-one operations across cluster boundaries the optimization is to first perform reductions locally within the clusters. This occurs, for example, when computing statistics (ATPG). Another technique to reduce intercluster traffic is caching at the cluster level. For example, in applications where duplicate data is sent in a (personalized) all-to-all exchange, intercluster exchanges can be coordinated by a single machine per cluster, making sure that the same data travels over the same WAN link only once. In Water this is used for reading and writing molecule data. Finally, our Red/Black nearest neighbor algorithm was rewritten to one with a more relaxed consistency, reducing the intercluster communication by dropping some row exchanges (SOR).

For RA and ASP we used forms of latency hiding. Asynchronous point-to-point messages allow message combining, which can also be applied at the cluster level. As in the cluster caching scheme, one processor per cluster is designated to handle all its intercluster traffic, only now the task is to combine messages to reduce overhead. The combined messages are sent asynchronously over the WAN, allowing the sending processor to overlap computation and communication (RA). As noted in Section 2, totally ordered broadcast performs badly on a multilevel cluster. The distributed sequencer implementation causes broadcasts to be limited by the speed of the wide-area link. For certain cases application behavior can be exploited. When several broadcasts are performed in a row by the same machine, they can be pipelined by using a single sequencer that is migrated to the machine that initiates the broadcasts (ASP). Furthermore, although we did not implement this, asynchronous broadcasts can be pipelined (ACP). Finally, in SOR there is room for latency hiding by using asynchronous point-to-point messages (not shown in the graphs).

The tables show that for the six applications where intercluster traffic reduction was used, traffic was indeed reduced, except for ATPG, where it was increased. For the other four applications, traffic was not greatly reduced, as was expected, for latency hiding techniques.

Except for RA, all applications run significantly faster on multiple clusters than on one cluster. It is surprising that what in retrospect looks like a few simple optimizations can be so good at masking two orders of magnitude difference in hardware performance in parts of the interconnect. Apparently, there is enough room in the algorithms to allow a limited number of LAN links to be changed into WAN links. Communication

Application	# RPC	RPC kbyte	# bcast	bcast kbyte
Water	25,665	56,826	3,000	102,919
IDA*	18,337	1,655	26,907	1,653
TSP	12,221	1,205	14,508	2,577
ATPG	3,451	206	1,659	76
SOR	1,443	4,395	5,199	260
ASP	111	976	38,139	53,171
ACP	1,105	10,323	112,911	35,790
RA	1,308,409	124,725	12,975	1,239

Table 4: Intercluster Traffic Before Optimization ($P = 64, C = 4$)

Application	# RPC	RPC kbyte	# bcast	bcast kbyte
Water'	4,609	5,179	3,165	102,929
IDA*'	6,461	708	31,239	2,065
TSP'	111	14	1734	495
ATPG'	573	481	3,531	656
SOR'	807	1,553	4,578	219
ASP'	313	959	9,558	52,407
ACP'	–	–	–	–
RA'	39,315	52,615	104,052	5,076

Table 5: Intercluster Traffic After Optimization ($P = 64, C = 4$)

smoothing and intercluster latency hiding can be thought of as exploiting slack in the algorithms; reduction of intercluster communication goes further and improves performance by restructuring the algorithm. As can be expected in a communication-hostile environment, intercluster traffic reduction achieves better results than latency hiding. It will be interesting to see if other optimizations for these and other algorithms exist, and whether they also fit into the same two categories.

6 Related Work

In this paper several medium grain algorithms (and optimizations) are studied on a multilevel communication network consisting of LAN and WAN links. The work is of direct relevance to research in meta computing. Projects such as Legion [19], Dome [2], MOL [31], Polder [28], Globe [21], Globus[17], Condor [15], VDCE [35], JavaParty [30], Java/DSM [38], and SuperWeb [1] aim at creating a software infrastructure—addressing such problems as job scheduling, heterogeneity, and fault tolerance—to support computing on a much larger scale than previously thought possible. So far, meta computing has been aimed at coarse grain, embarrassingly parallel jobs (see, for example [18]). Our work studies applications with more challenging communication behavior.

Several other researchers have worked on performance analyses of parallel algorithms on specific architectures. Martin et al. [27] perform an analysis of the sensitivity of applications to changes in the LogP parameters in a single-level Myrinet cluster, by putting a delay loop on the Myrinet cards. A number of papers study algorithm behavior on NUMA machines, most notably, the papers on the Splash benchmark suite [33, 37]. NUMA latencies differ typically by a factor of 2 to 3, and directory based invalidation cache coherency achieves adequate performance for a wide range of applications [10, 26]. In wide-area multilevel clusters latency and bandwidth differences are much larger, and we used various forms of algorithm restructuring to achieve adequate performance.

A number of researchers are working on coherency models for clustered SMPs. Here, the latency difference is typically one order of magnitude—larger than NUMA, smaller than a LAN/WAN cluster. Stets et al. [34] present "moderately lazy" release consistency with a global directory in Cashmere-2L (for two-level), to allow a higher level of asynchrony. Bilas et al. [6] study the behavior of a novel coherency scheme in a cluster of shared memory machines (SMPs). One of their conclusions is that for good performance algorithm restructuring is necessary, as we have done here. Other notable papers on distributed shared memory for clustered SMPs are on TreadMarks [13], Shasta [32], and SoftFLASH [16].

Wide area networks, multilevel communication structures, and performance of challenging applications, have all been studied before. Already, some applications have been ported to meta computing environments. Our study takes a logical next step and combines and extends insights to analyze performance and to suggest ways to adapt several commonly encountered communication structures to run well on a multilevel LAN/WAN cluster.

Looking into the future, we expect multilevel communication structures with a wide performance range to become more prevalent. Going from the large to the small, Gigabit networking and meta computing initiatives will make it increasingly attractive to cluster LANs over WANs [11]; small scale SMPs will be in-

creasingly clustered in LANs [14]; and finally, when multiple processors on a single chip arrive, SMPs can be constructed out of these, to create even more processing power [20]. It seems likely that differences in bandwidth and latency will increase, providing challenges for algorithm designers and systems researchers.

7 Conclusion

Recent technology trends have created a growing interest in wide-area computing. The limited bandwidth and high latency of WANs, however, make it difficult to achieve good application performance. Our work analyzes several nontrivial algorithms on a multilevel communication structure (LAN clusters connected by a WAN). Both the bandwidth and the latency of the LAN and WAN links differ by almost two orders of magnitude. Several optimization techniques were used to improve the performance of the parallel algorithms.

We have analyzed the applications, and found that, as expected, most existing parallel applications perform worse on multiple clusters connected by a WAN than on a single LAN cluster with the same number of processors. Some applications even perform worse on two (or four) clusters of 16 (or 15) processors than on a single cluster. Since the applications had been designed and tuned for a single cluster, it is not surprising that performance suffers when some of the links become much slower. It is surprising, however, to see that the optimizations that we subsequently implemented worked so well. The optimizations either reduce intercluster traffic or mask the effect of intercluster communication. In some cases the algorithm was restructured, in others the implementation of communication primitives was refined.

Most of the optimization techniques work across clusters, and do not improve performance within a single cluster. They can, however, be viewed as adaptations of general communication reduction and latency hiding techniques, such as load balancing, caching, distributed reduction operators, pipelining, and relaxing the data consistency requirements of an algorithm. It is encouraging to see that these well-known ideas are apparently general enough to uncover enough exploitable communication patterns to achieve acceptable performance on a wide-area system. On the other hand, it is remarkable that each optimization is used in only one application. This is in contrast with the work on NUMA machines where a single technique is used for all applications. Given the much larger performance difference between local and remote communication in our system, it was to be expected, perhaps, that we had to resort to algorithm restructuring.

There is clearly a limit to what can be achieved when local clusters are connected through wide area networks. Ultimately, the inherent communication of an algorithm and the limited bandwidth and high latencies of wide area links limit performance. Ingenuity in devising novel intercluster traffic reduction and latency hiding techniques can only go as far as the grain of the algorithm, physics, and the state of the art in networking technology allow—as retrograde analysis illustrates, since here intercluster communication could not be reduced enough to allow acceptable speedup.

Nevertheless, our conclusion is that many medium grain applications can be optimized to run successfully on a multilevel, wide-area cluster. The conclusion makes the work on meta computing all the more valuable, since a wider class of algorithms can be run on them than previously thought.

Future research in this area can look for more optimizations for more applications. Performance was found to be quite sensitive to problem size, number of processors, number of clusters, and latency and band-

width. This paper has only scratched the surface of these intricate issues, and further sensitivity analysis is part of our future work. Where the optimizations are instances of more general techniques, they can be used in wide-area parallel programming systems.

Acknowledgments

We gratefully acknowledge the support of the Netherlands Organization for Scientific Research (NWO) and the ASCI research school for making the DAS project possible. We thank the Orca and the Albatross group for valuable feedback and criticism. Several people did a great deal of work to make this research possible. In particular, we thank Kees Verstoep and Ceriel Jacobs. We thank Rien van Veldhuizen for his help with numerical convergence in SOR and suggesting the relation of our scheme to chaotic relaxation. Dick Grune, Ceriel Jacobs, Koen Langendoen, John Romein, Andy Tanenbaum, and Kees Verstoep gave useful comments on the paper.

References

[1] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. SuperWeb: Research Issues in Java-Based Global Computing. *Concurrency: Practice and Experience*, 9(6):535–553, June 1997.

[2] J. Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan. Dome: Parallel programming in a heterogeneous multi-user environment. In *10th International Parallel Processing Symposium*, pages 218–224, April 1996.

[3] H.E. Bal and L.V. Allis. Parallel Retrograde Analysis on a Distributed System. In *Supercomputing '95*, December 1995. Online at http://www.supercomp.org/sc95/proceedings/.

[4] H.E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Rühl, and K. Verstoep. Performance of a High-Level Parallel Language on a High-Speed Network. *Journal of Parallel and Distributed Computing*, 40(1):49–64, February 1997.

[5] H.E. Bal, A. Plaat, M.G. Bakker, P. Dozy, and R.F.H. Hofman. Optimizing Parallel Applications for Wide-Area Clusters. Technical Report IR-430, Vrije Universiteit Amsterdam, September 1997. Available from http://www.cs.vu.nl/albatross/.

[6] A. Bilas, L. Iftode, and J.P. Singh. Shared Virtual Memory across SMP Nodes using Automatic Update: Protocols and Performance. Technical Report TR-96-517, Princeton University, 1996.

[7] G.E. Blelloch, P.B. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. In *Proc. 7th ACM Symp. Par. Alg. and Arch. (SPAA)*, pages 1–12, July 1995.

[8] R.D. Blumofe, M. Frigo, C.F. Joerg, C.E. Leiserson, and K.H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proc. 8th ACM Symp. Par. Alg. and Arch. (SPAA)*, June 1996.

[9] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W. Su. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[10] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal. Directory-Based Cache Coherence in Large-Scale Multiprocessors. *IEEE Computer*, 23(6):49–58, June 1990.

[11] S. Chatterjee. Requirements for Success in Gigabit Networking. *Communications of the ACM*, 40(7):64–73, July 1997.

[12] D. Chazan and W. Miranker. Chaotic relaxation. *Linear Algebra and its Applications*, 2:199–222, 1969.

[13] A. Cox, S. Dwarkadas, P. Keheler, H. Lu, R. Rajamony, and W. Zwaenepoel. Software versus hardware shared-memory implementation: a case study. In *Proc. 21st Intern. Symp. Comp. Arch.*, pages 106–117, April 1994.

[14] D. E. Culler and J. P. Singh with A. Gupta. *Parallel Computer Architecture—A hardware/software approach.* Morgan Kaufman, 1997. preprint available from http://http.cs.berkeley.edu/~culler/book.alpha/index.html.

[15] D.H.J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A Worldwide Flock of Condors: Load Sharing among Workstation Clusters. *Future Generation Computer Systems*, 12(1):53–66, May 1996.

[16] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy. SoftFLASH: Analyzing the performance of clustered distributed virtual shared memory. In *Proc. 7th Intern. Conf. on Arch. Support for Prog. Lang. and Oper Systems*, pages 210–220, October 1996.

[17] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Int. Journal of Supercomputer Applications*, 11(2):115–128, Summer 1997.

[18] A.S. Grimshaw, A. Nguyen-Tuong, and W.A. Wulf. Campus-Wide Computing: Results Using Legion at the University of Virginia. Technical Report CS-95-19, University of Virginia, March 1995.

[19] A.S. Grimshaw and Wm. A. Wulf. The Legion Vision of a Worldwide Virtual Computer. *Comm. ACM*, 40(1):39–45, January 1997.

[20] L. Hammond, B.A. Nayfeh, and K. Olukotun. A Single-Chip Multiprocessor. *IEEE Computer*, September 1997. Special Issue on Billion-Transistor Processors.

[21] P. Homburg, M. van Steen, and A.S. Tanenbaum. Communication in GLOBE: An Object-Based Worldwide Operating System. In *Proc. Fifth International Workshop on Object Orientation in Operating Systems*, pages 43–47, October 1996.

[22] R.H. Klenke, R.D. Williams, and J.H. Aylor. Parallel-Processing Techniques for Automatic Test Pattern Generation. *IEEE Computer*, 25(1):71–84, January 1992.

[23] Richard E. Korf. Iterative Deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[24] K.G. Langendoen, R. Hofman, and H.E. Bal. Challenging Applications on Fast Networks. Technical report, Vrije Universiteit, July 1997.

[25] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *24th Ann. Int. Symp. on Computer Architecture*, pages 241–251, June 1997.

[26] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M.S. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[27] R.P. Martin, A.M. Vahdat, D.E. Culler, and T.E. Anderson. Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture. In *24th Ann. Int. Symp. on Computer Architecture*, pages 85–97, June 1997.

[28] B. J. Overeinder, P. M. A. Sloot, R. N. Heederik, and L. O. Hertzberger. A Dynamic Load Balancing System for Parallel Cluster Computing. *Future Generation Computer Systems*, 12:101–115, May 1996.

[29] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Supercomputing '95*, San Diego, CA, December 1995.

[30] M. Philippsen and M. Zenger. JavaParty—Transparent Remote Objects in Java. In *ACM 1997 PPoPP Workshop on Java for Science and Engineering Computation*, June 1997.

[31] A. Reinefeld, R. Baraglia, T. Decker, J. Gehring, D. Laforenza, J. Simon, T. Rümke, and F. Ramme. The MOL Project: An Open Extensible Metacomputer. In *Heterogenous computing workshop HCW'97 at IPPS'97*, April 1997.

[32] D. J. Scales, K. Gharachorloo, and A. Aggarwal. Fine-grain software distributed shared memory on SMP clusters. Technical Report WRL 97/3, DEC Western Research Laboratory, February 1997.

[33] J.P. Singh, W-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *ACM Computer Architecture News*, 20(1):5–44, March 1992.

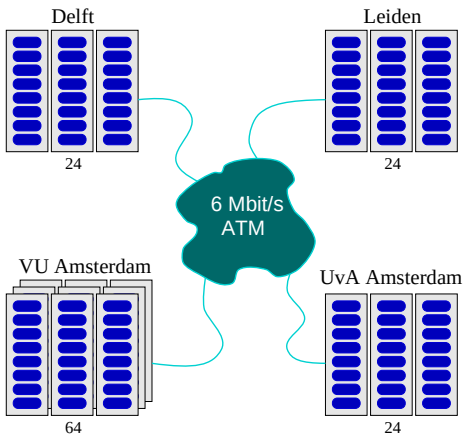
[34] R. Stets, S Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software coherent shared memory on a clustered remote-write network. In *Proc. 16th ACM Symp. on Oper. Systems Princ.*, October 1997.

[35] H. Topcuoglu and S. Hariri. A Global Computing Environment for Networked Resources. In *Proc. 1997 Int. Conf. on Parallel Processing*, pages 493–496, Bloomingdale, IL, August 1997.

[36] M. Welsh, A. Basu, and T. von Eicken. ATM and Fast Ethernet Network Interfaces for User-level Communication. In *Proceedings of the Third International Symposium on High Performance Computer Architecture*, pages 332–342, San Antonio, Texas, February 1997.

[37] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.

[38] W. Yu and A. Cox. Java/DSM: A Platform for Heterogeneous Computing. In *ACM 1997 PPoPP Workshop on Java for Science and Engineering Computation*, June 1997.



ıre 17: The Distributed ASCI Supercomputer