Applications

Parallel N-Body Simulation on a Large-Scale Homogeneous Distributed System *

John W. Romein and Henri E. Bal

Vrije Universiteit, Department of Mathematics and Computer Science,
Amsterdam, The Netherlands, {*john,bal*}*@cs.vu.nl*

Abstract. A processor pool is a homogeneous collection of processors that are used for computationally intensive tasks, such as parallel programs. Processor pools are far less expensive than multicomputers and more convenient to use than collections of idle workstations. This paper gives a case study in parallel programming on a processor pool with 80 SPARCs connected by an Ethernet, running the Amoeba distributed operating system. We use a realistic application (N-body simulation of water molecules) and show that a good performance can be obtained. We measured a speedup of 72 on 80 processors.

1 Introduction

Networks of idle workstations are often used for running parallel applications. Many institutes have a distributed computing system consisting of many powerful workstations, which are idle most of the time. Using this (otherwise wasted) processing power for running coarse-grained parallel applications thus seems attractive.

Unfortunately, this approach to parallel processing also has several disadvantages, which are caused by the heterogeneous nature of such distributed systems. Most institutes have different types of workstations running different operating systems, which were not designed for parallel programming. The workstations usually run dozens of daemon processes that occasionally become active, so the workstations are not really idle. The machines are typically connected by various networks, which are used heavily for file transfers and other applications.

These factors make it difficult to obtain good performance for parallel programs that run on collections of workstations. Load balancing, for example, becomes very difficult on such systems, because processors may run at different speeds or may not have all CPU cycles available for the parallel program. Also, the performance measurements are obscured by many external factors (e.g., congestion of the network due to large file transfers or page swapping) and thus are hard to reproduce.

The problems described above do not occur with multicomputers, since these are usually constructed out of identical processors which all run a single operating system. Also, the interconnection network on modern multicomputers is orders of a magnitude faster than current local area networks, thus allowing more fine-grained parallelism.

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

Unfortunately, multicomputers are very expensive, whereas using idle workstations essentially is for free.

An attractive alternative to collections of workstations and multicomputers is the *processor pool model* [8]. A processor pool is a homogeneous collection of processors shared among all users of a distributed system. The processors are used for computationally intensive tasks. At any moment, the pool can run many applications from different users, a parallel application from one user, or a combination of these. The processors are connected by a local area network and do not have any peripheral devices (like displays or disks), thus making them very cheap.

In this paper, we present a case study in parallel processing on a fairly large-scale processor pool. We will show that excellent and reproducible speedups can be obtained for a realistic application. Since processor pools are far less expensive than multicomputers, this case study indicates that they are a feasible platform for parallel processing. Another contribution of this paper is a description of several optimizations that can be applied to such applications, to reduce the communication overhead.

The paper is organized as follows. In Sect. 2, we describe the computing environment we use. The hardware consists of 80 SPARC processors connected by an Ethernet and runs the Amoeba distributed operating system. Next, in Sect. 3, we discuss the application that we have used in our case study. The application is one of the SPLASH [7] applications: the simulation of water molecules. The original N-body simulation program was written for shared memory machines. We have rewritten the program to run on a distributed system. Section 4 describes our distributed water simulation program. Next, in Sect. 5 we study the performance of the program on the Amoeba processor pool and show that high speedups can be obtained. In Sect. 6, we give some conclusions and we look at related work.

2 The Amoeba Processor Pool

This section describes our computing environment. The hardware is described in Sect. 2.1 and the Amoeba operating system is discussed in Sect. 2.2.

2.1 The Processor Pool

The processors in the pool are connected by a 10 Mbit/s Ethernet. To increase the bandwidth of the network, multiple Ethernet segments are used, which are connected by a low-cost switch. There are ten segments of eight processors each. Some special servers (e.g., file servers) are on separate segments.

The segments are connected by a Kalpana switch. The switch forwards Ethernet packets between segments as follows:

- A packet with both the source and the destination on the same segment is not forwarded to other segments.
- A packet with the source and destination on different segments is forwarded with a very small delay (40 μs). The overhead of the switch is negligible compared to the latency of Ethernet packets. The switch uses a crossbar architecture. The advantage

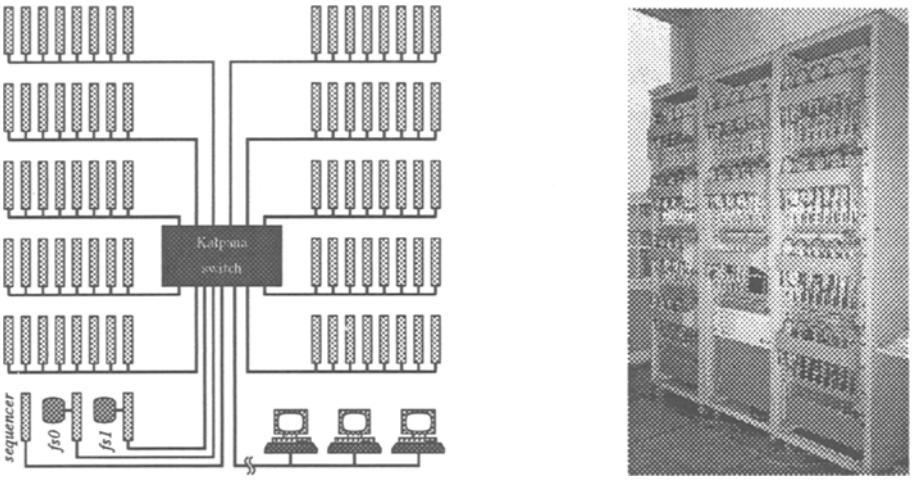

Fig. 1. The Amoeba processor pool.

of using a switch is that multiple pairs of processors can communicate at the same time, provided that the messages do not conflict. Packets sent to the same segment do conflict; in this case the switch can buffer up to 256 packets.

- A multicast packet is forwarded with about the same delay as a unicast packet to all other segments.

The processor boards are SPARCclassic clones made by Tatung. Each board is equipped with one 50 MHz Texas Instruments V8-SPARC processor (a TMS390S10). All processor pool boards have 32 Mb RAM. In addition there is one processor board that has 128 Mb RAM. This board is used by the sequencer [4] of Amoeba's multicast protocol.

The processor pool can be regarded as a low-cost multicomputer. (Its total cost, including 80 SPARCs, 2.5 Gb memory, the network, switch, file servers, sequencer, and packaging is about US$ 350,000). Alternatively, the machine can be regarded as an idealized version of the idle workstations model. It has the same architecture as a homogeneous collection of workstations on a network.

2.2 Amoeba

The operating system we use is Amoeba [8]. Amoeba is designed with the processor pool model in mind. It assumes that the system contains a large number of processors, so all servers (e.g., file server, directory server) run on separate processors, as user processes. User programs are run on the processor pool, using as many processors as the application requires. One of the main advantages of this software organization is that idle processors on Amoeba are truly idle and do not run any daemon processes.

Amoeba is a capability-based microkernel, which basically provides processes, kernel-level threads, synchronization mechanisms, and communication protocols. There are two high-level communication protocols: *Group Communication* [4] and *Remote Procedure Calls* [1].

Reliable communication between two processes is done with the Remote Procedure Call (RPC) primitives `getreq`, `putrep`, and `trans`. A server thread that wants to accept an RPC request calls `getreq` and blocks until a request arrives. Client threads can invoke remote procedures with the `trans` primitive, which blocks until the server thread returns a reply message using `putrep`.

Group communication is supported with multicast messages. A process can send a message to all members of a group using the `grp_send` primitive. The Amoeba kernel buffers the messages until the application does a `grp_receive`. The protocol is reliable and totally-ordered, which means that all machines receive all messages in the same order [4]. The implementation is efficient. A `grp_send` merely costs two low level messages in the normal case, independent of the number of receivers.

3 The 'Water' Program

We have used an N-body simulation program as a case study for programming the Amoeba processor pool. The program, called *Water*, simulates a system in which H_2O molecules move in an imaginary box. The molecules interact with each other by gravitational forces, and the atoms within a molecule also interact with each other.

The computation is repeated over a user-specified number of timesteps. During a timestep (in the order of femto-seconds = 10^{-15} s), the Newtonian equations of motion for the water molecules are set up and solved using Gear's sixth-order predictor-corrector method [3]. The forces between and within the molecules are computed, and the total potential energy of the system is calculated. The box size is large enough to hold all molecules. Physicists use the program to predict a variety of static and dynamic properties of water in liquid state.

The original *Water* program was written in FORTRAN and is included in the Perfect Club [2] set of supercomputing benchmarks. It is completely sequential. The SPLASH [7] version is a rewrite of the sequential FORTRAN code to a parallel C program. This version is designed to run on shared memory multiprocessors.

We have reimplemented the SPLASH version (again in C) to run on a distributed architecture without a shared address space. This program is a complete reimplementation, because many changes were needed to have it run efficiently on a distributed machine. Our implementation uses explicit message passing (RPC and multicast).

A major issue in implementing a program like *Water* on a processor pool is how to reduce the communication overhead. The program potentially sends a very large number of messages over the Ethernet. We have addressed this problem by carefully analyzing the communication behavior of the program and by optimizing the most important cases, as will be discussed in Sect. 4.

Although the three *Water* implementations are quite different from each other, they share the same program structure, as shown in Fig. 2. The statements shown in bold face are only used in our distributed program.

During initialization, the program reads the input data, which contains the number of molecules that should be simulated, their initial positions and velocities, the length of one timestep, how many timesteps the program should simulate, and how often the results

```
int main()
{
    Set up scaling factors and constants;
    Read positions and velocities;
    Estimate accelerations;
    Send updates of accelerations;

    while (compute another timestep)
    {
        Compute predictions;
        Multicast positions;
        Compute forces within a molecule;
        Compute forces between molecules;
        Send updates of forces;
        Correct predictions;
        Put molecules back inside the box if they moved out;
        Compute kinetic energy;

        if (print results of this timestep)
        {
            Multicast positions;
            Compute potential energy;
            Use tree RPC to compute shared sum;
            Print results;
        }
    }
}
```

Fig. 2. Main loop of the *Water* program.

should be printed. Then it computes the boxsize, some input-dependent and -independent constants, and finally it does some implementation-dependent initializations.

After initialization, the program repeatedly performs the predictions, computes intra- and intermolecular forces, the kinetic energy, places molecules that moved out of the box back inside the box, and corrects the predictions. If the results of that timestep should be printed, the potential energy of the system is computed and displayed.

4 A Distributed 'Water' Program

In this section, we will describe how we developed a distributed *Water* program that runs efficiently on the Amoeba processor pool. In general, porting a program from a shared memory machine to a distributed memory machine requires the following changes:

- The shared data must be partitioned over the available machines, or replicated on several machines. This requires a careful study of the way in which the processes access the shared data.
- Synchronization and access to remote memories must be done by passing messages.
- The parallel algorithm must be adapted such that the number of messages sent across the network is minimal.

We carefully studied the SPLASH implementation, but ultimately decided to rewrite almost all of it (except for some code computing mathematical formulas). The main rea-

son for rewriting the program was to obtain an efficient implementation on a distributed system. We have determined several important optimizations that reduce the communication overhead significantly. In the following subsections, we will discuss the work distribution and data distribution in the program, and we will analyze the complexity of the computations and communication.

4.1 Work Distribution

Both the SPLASH program and the distributed program achieve work distribution by partitioning the molecules over the available processors. Each processor 'owns' a number of molecules and does all computations needed for that molecule. The distribution is done statically. As soon as the input is read (and the number of molecules is known) the molecules are assigned to the processors. No provision is made to dynamically balance the load of each processor. Dynamic load balancing would be very hard to implement efficiently in this application.

4.2 Data Distribution

Apart from some shared sums that are needed for the computation of the potential energy, the most important data structure in the distributed memory program is a large four-dimensional array. This array holds the data for all molecules. Figure 3 shows the declarations of the shared data in simplified form.

```
#define MAX_MOLECULES 4096
#define MAX_ORDER        8 /* POS, VEL, ACC, 3rd .. 6th derivatives, FORCES */
#define NR_ATOMS         3 /* H1, O, H2 */
#define NR_DIRS          3 /* X, Y, Z */

double data [MAX_MOLECULES] [MAX_ORDER] [NR_ATOMS] [NR_DIRS];


#define NR_SHARED_SUMS   6

double potential_energy [NR_SHARED_SUMS];
```

Fig. 3. Shared data declarations.

For each molecule, the *position*, *velocity*, *acceleration*, the $3^{rd} \ldots 6^{th}$ *derivatives*, and the *forces* are obtained through the second index value, for *each atom* and in *each direction*.

Each processor allocates storage for the entire array, although this is not strictly necessary, since only a small part of the array is really used. However, even with thousands of molecules only a few megabytes of memory are needed, so this allocation scheme did not turn out to be a problem. Efficient access to the data is more important.

To illustrate how the program distributes the data of the molecules, consider Fig. 4, which shows an example with 12 molecules and 4 processors. Each processor is the

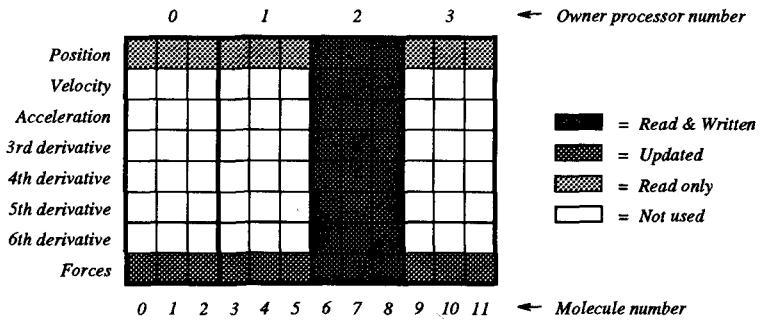

Fig. 4. Example of data distribution on processor 2 if $p = 4$ and $n = 12$.

owner of $\frac{12}{4} = 3$ molecules. Only the home processor of a molecule has all the molecule's data available. The positions and forces are implemented as described below. The remaining pieces of information are private to the owner.

The *positions* of the molecules (or rather the H and O atoms within the molecules) are replicated everywhere. After being computed, the positions are read many times by many processors, so it is evident that they should be replicated to obtain good performance.

The *forces* acting on the molecules are updated (i.e. incremented) very frequently by all processors. As we will describe in Sect. 4.5, updating of the forces can be implemented efficiently by having each processor first compute local sums of partial forces and then communicate the final values of these local sums.

The program tries to overlap communication and computation as much as possible. If a processor knows that other processors will need its data, these data are sent in advance, as soon as they are available. The receiver picks up a message (or waits for it) when it needs the data. This is more efficient than to let the processor that needs the data initiate a request for the data followed by a reply from the supplier. The way it is implemented now, synchronization comes for free: one message is used for both synchronization and shipping data.

In the next three subsections, we discuss three aspects of data distribution in our program: the replication of positions, the implementation of partial sums, and the updating of remote forces. Finally, we will give a complexity analysis of the communication and computation times.

4.3 Replication of Positions Using Multicast Messages

Figure 4 shows that the positions of all molecules are replicated everywhere. As these molecules move in the box, their positions have to be updated sometimes. This happens at two synchronization points (see also Fig. 2):

- after the predictions and before the computation of the inter-molecular forces,
- if the results of the current timestep should be printed, the positions of the molecules are needed for the computation of the potential energy.

To update replicated positions, we use the Amoeba multicast primitives. Each processor marshalls all positions of the atoms in the molecules that it owns into one message and multicasts it to all processors. Next, the processor receives one message from each other processor, containing the positions of the other molecules' atoms. If there are n molecules and p processors, each message contains updates for $\frac{n}{p}$ molecules and for each molecule nine doubles are needed (three atoms, H, O and H, in three directions), so a message contains $72\frac{n}{p}$ bytes of data. In addition, each message has 8 bytes of fixed information, so the total number of bytes transferred over the network is $72n + 8p$. This number excludes the headers prepended by the various network software layers.

4.4 Implementing Shared Sums

A different kind of synchronization is needed for the computation of the potential energy. The total potential energy is the sum of some partial values computed by the individual processors. Processor 0 prints the total potential energy, so only this processor needs to know the total. In fact, six shared sums are computed at the same time.

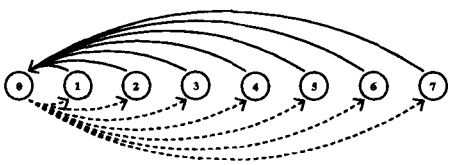

Fig. 5. Naive implementation of computing a shared sum.

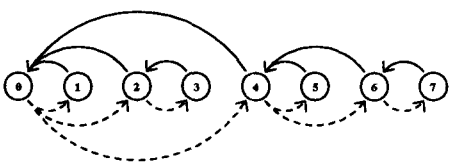

Fig. 6. Indirect RPCs in a conceptual tree.

A naive way to implement this is to let each processor send an RPC message to processor 0 containing a partial value, and let processor 0 add those values together (see Fig. 5). The request messages, indicated by a solid line, carry those values. The reply messages, drawn with a dashed line, are empty. This means that processor 0 receives $p - 1$ subsequent RPCs, surely becoming a bottleneck if p is large.

Figure 6 shows a much more efficient implementation, which avoids the bottleneck. Informally, processors receive some RPCs with partial values, add these values together and initiate an RPC to another processor, using a conceptual tree-structure. Processor 0 eventually receives some values and adds these together to obtain the final sum. Most RPCs can be done simultaneously, and no processor receives more than $\lceil \log p \rceil$ messages. We have used this scheme in our program.

4.5 Updating Remote Forces

Computation of the inter-molecular forces is done by computing the pair-wise interactions between molecules. This is an $\mathcal{O}(\frac{n^2}{p})$ problem. If the distance between a pair is *less* than half the boxsize, the force between the molecules is computed by the owner processor of one of them. Otherwise, the force is not computed but neglected. If the other molecule resides on another processor, communication will be necessary.

To compute the force between two molecules one needs the positions of both molecules. Positions are fully replicated, so they are available on the processor that does the computation. Once the force is computed, it has to be added to the total force of both molecules. One of the molecules resides on the processor where the computation is done, so the force can be added immediately. This cannot be done for the other molecule, because it may be on another processor, so we need to do communication here.

The total force acting on a molecule is the sum of partial forces. Each molecule receives updates from approximately half the other molecules, and sends updates to the other half. To reduce communication overhead, we combine updates to several molecules into one RPC. Each processor needs to send an RPC to approximately half the other processors, so the total number of RPCs approximates $p\frac{p}{2} = \mathcal{O}(p^2)$. Each RPC contains updates for $\frac{n}{p}$ molecules, which takes about $72\frac{n}{p}$ bytes. Hence, a total of $36np$ bytes are sent over the network.

4.6 Complexity Analysis of Communication and Computation Time

Before presenting performance measurements, we will first discuss the complexity of the communication and the computations done by the *Water* program. Figure 7 summarizes the total number of messages and the total number of bytes sent during one timestep, as discussed in the last three subsections.

Sync. point	# RPCs	# multicasts	# bytes
Multicast positions	–	p	$72n + 8p$
Update forces	$\frac{p^2}{2}$	–	$36np$
Multicast positions	–	p	$72n + 8p$
Update potential energy	$p - 1$	–	$48(p - 1)$
Total	$\frac{p^2}{2} + p - 1$	$2p$	$36np + 144n + 64p - 48$

Fig. 7. Total number of RPCs, multicasts, and bytes sent per timestep.

With regard to computation time, the most time consuming part of the program is the routine where the inter-molecular forces are computed. For each molecule owned by some processor, the interactions with half the other molecules are computed, thus the complexity for this computation is $\mathcal{O}(\frac{n^2}{p})$.

The amount of time that is needed for the communication is $\mathcal{O}(np)$. If we compare this to the computational complexity, $\mathcal{O}(\frac{n^2}{p})$, we see that the ratio between computation and communication time will be high if n is large. It is important to maintain tens of molecules per processor to obtain reasonable speedups.

5 Performance Measurements

The measurements of the distributed program were done on the Amoeba processor pool described in Sect. 2. All processors ran an Amoeba 5.2 kernel. The application was

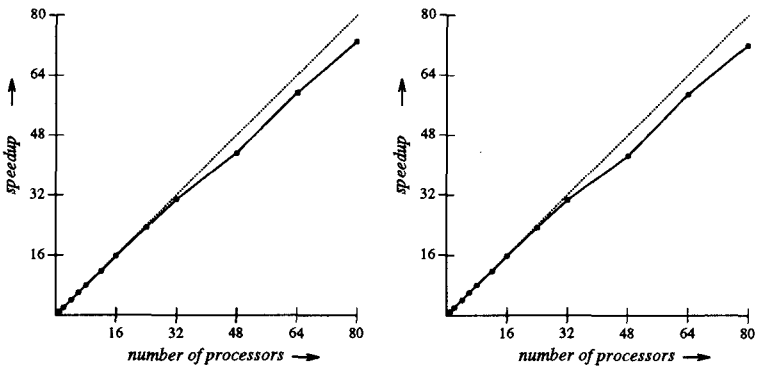

Fig. 8. Speedups obtained on up to 80 processors. The left figure shows speedups without computation of the potential energy, the right figure shows the speedups for the second timestep.

p	# RPCs	# multicasts	# bytes	# RPCs	# multicasts	# bytes
	Theoretical values			*Measured values*		
2	3	4	884816	3	4	884816
80	3279	160	12391376	3317	160	12530136

Fig. 9. Theoretical and measured values for $n = 4096$ molecules.

compiled with gcc 2.5.8 with the -O2 -mv8 options. An input file for 4096 molecules was generated. The molecules were placed in a 16 × 16 × 16 box and initialized with random velocities.

During each run two values are measured. First, the execution time of the second timestep without the computation of the potential energy was measured (as in the SPLASH paper). They did not measure the first timestep, to avoid cold-start cache misses. Second, to get a more realistic impression of the total speedup, the execution time of the entire second timestep was measured. The speedups for the program (relative to the single CPU case) are shown in Fig. 8.

On a single SPARC processor, the entire second timestep takes 7451.9 seconds. On 80 processors, it takes about 103.5 seconds, so a 72 fold speedup is obtained. If the computation of the potential energy is omitted, the speedup is even 73. The efficiency is similar to that obtained by the original SPLASH implementation on a shared-memory multiprocessor, although we use a larger problem size (4096 molecules instead of 288).

The execution times are reproducible. The differences in execution times generally are well below 1%. In one case we measured a difference of 4.1%.

To analyze the performance, we have looked at the RPC and multicast messages generated by the program. Figure 9 shows the theoretical and measured values for the total number of RPCs, multicast messages, and bytes sent on 2 and 80 processors using 4096 molecules. The theoretical values are based on the formulas in Fig. 7. For 80 processors the measured values differ somewhat, because p is not a power of 2 and n is not a multiple of 80, leading to an uneven distribution of the molecules.

It is also interesting to see how large the messages are. With 80 processors, each

multicast message and most (97.6%) RPC requests are about 3.6 Kb. All RPC replies are null-messages. Large messages are split into several Ethernet packets.

As an advantage over an architecture with a single Ethernet segment, the Kalpana switch allows simultaneous message delivery in two different ways (see Sect. 2.1). First, the majority of the messages (a little under 90%) are unicast messages that must be forwarded from one segment to another. As long as the destinations are not on the same segment, different messages can be delivered in parallel. Second, about 10% of the messages are unicast messages with the source and destination on the same segment. The switch does not forward these messages, so the other segments are still available for other messages. Finally, 2.4% of the messages are multicast messages that must be forwarded to all other segments. This involves all segments, so different multicast messages cannot be sent simultaneously.

For 80 processors, about 12.5 Mb of data are sent in a little more than 100 seconds. On average, each processor sends about 160 Kb of data. Since the Ethernet has a bandwidth of 10 Mbit/sec, each processor in theory spends only a small amount of time (less than a second) doing communication. However, the communication is very bursty. The network is idle for a long time and then all processors start to send data at the same moment. We have measured that the delays for sending and receiving data at the different synchronization points can be as high as 7 seconds, so the network still is a limiting factor.

Besides the network, another reason why the speedup is less than 80 is the fact that the load is not balanced optimally. With 4096 molecules and 80 processors, some processors are assigned 51 molecules and others get 52 molecules. This leads to a small load imbalance.

We also implemented two alternative schemes for updating the remote forces [6], but measurements showed that the scheme described above yields better speedups and absolute execution times, at least with 80 processors and 4096 molecules.

6 Conclusions and Related Work

We have implemented a parallel N-body simulation program on the Amoeba processor pool. We have used the SPLASH shared-memory program as a starting point, but we rewrote almost all of it. The reason for rewriting it was that several optimizations were necessary to reduce the communication overhead. The result (for a large input problem) is a speedup of 72 on 80 processors. We think this speedup is excellent, given the fact that we use a slow and unreliable network (Ethernet) and fast, modern processors (50 MHz SPARCs).

The experiment thus has shown that good performance can be obtained for a realistic application on the processor pool. The main advantage of the processor pool model over the idle workstations model lies in the possibility to allocate the processors and the network segments exclusively to one (parallel) application. The program will not be slowed down by other activities on the processors or network, such as daemon processes or file transfers. Compared with a multicomputer, the processor pool is far less expensive. As should be clear from the previous sections, however, porting a program like *Water* from a shared memory machine to a distributed system requires a significant effort.

The SPLASH program has also been ported to Jade [5]. Jade is a high-level parallel programming language for managing coarse grained concurrency. The C language is extended to provide the programmer a shared address space and the idea of sequential programming. A Jade programmer provides information about how shared data are accessed. The paper shows several results with their implementation of *Water* running on different parallel architectures. The architecture that looks most like ours consists of 32 SPARC ELCs, connected by a 10 Mbit/s Ethernet. They obtain a speedup of a factor 12 using 24 processors; with more processors the speedup decreases. Using other architectures they obtain considerably better speedups. The reason they get inferior speedups is that they try to automate the communication for fine grained access to distributed shared memory, while we hand-optimized the accesses to global data to minimize the total amount of communication needed.

In conclusion, our case study shows that the processor pool model is a feasible platform for parallel processing. A processor pool is far less expensive than a multicomputer. One disadvantage (shared with the idle workstations model) is the fact that the network interconnecting the processors is slow. We have shown, however, that excellent speedups still can be obtained for a realistic problem. On modern networks with higher bandwidths (such as ATM), one can expect even better performance.

Acknowledgements

Saniya Ben Hassen, Raoul Bhoedjang, Dick Grune, and Koen Langendoen provided useful comments on a draft of this paper.

References

1. A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.
2. M. Berry et. al. The perfect club benchmarks: Effective performance evaluation of supercomputers. Technical Report CSRD Report No. 827, Center for Supercomputing Research and Development, Urbana, Illinois, May 1989.
3. C.W. Gear. *Numerical Initial Value Problems in Ordinary Differential Equatations*. Prentice-Hall, New Jersey, 1971.
4. M.F. Kaashoek. *Group Communication in Distributed Computer Systems*. PhD thesis, Vrije Universiteit, Amsterdam, December 1992.
5. M.C. Rinard, D.J. Scales, and M.S. Lam. Jade: A high-level, machine-independent language for parallel programming. *IEEE Computer*, 26(6):28–38, June 1993.
6. J.W. Romein. Water — an N-body simulation program on a distributed architecture. Master's thesis, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, August 1994.
7. J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. *ACM Computer Architecture News*, 20(1), March 1992.
8. A.S. Tanenbaum, R. van Renesse, H. van Staveren, G.J. Sharp, S.J. Mullender, A.J. Jansen, and G. van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, December 1990.