The Parform — A High Performance Platform for Parallel Computing in a Distributed Workstation Environment

Clemens H. Cap, Volker Strumpen*
{cap, strumpen}@ifi.unizh.ch
Institut für Informatik
Universität Zürich
Winterthurerstrasse 190
CH-8057 Zürich, Switzerland

June 23, 1992

Abstract

The typical workstation in a LAN is idle during large periods of time. Under the concept of a *hypercomputer* this unused, distributed computing power can be put at the disposal of the user. The dynamic, heterogeneous, and distributed environment calls for a platform taking care of transparency, parallelization, load balancing and other issues. We describe such a system which, by optimized design and dynamic load distribution, proves faster than many related approaches. Performance measurements and an analytic model on scalability are presented for an explicit finite difference PDE solver.

Keywords: Distributed parallel computing, idle workstations, dynamic load balancing, scalability, finite difference methods.

Subject Classification: D.1.3, D.4.7, C.2.5 (ACM/CRCS)

1 Introduction

Present trends in supercomputer development emphasize expensive technologies and specialized architectural concepts. Furthermore, we observe a significant increase of

*Supported by *Siemens AG, ZFE, Germany* and *Schweizer Bundesamt für Konjunkturfragen*, Grant No. 2255.1

workstation performance and communication bandwidth, together with a shift of market interest from mainframes to workstations. Networks of high performance workstations become increasingly available in companies and research institutions. For a large percentage of their lifetime the machines are merely used for reading email, editing files and similar small tasks. However, a workstation network may also be considered as a virtual supercomputer or *hypercomputer* offering parallel programming features. In research institutions with state of the art RISC workstations often some 300 Mflops are wasted by idling. A number of recent research activities try to exploit the computing power of such environments, for example LINDA [1] and PVM [2].

In this paper we identify and describe the main problems of those approaches. Section 2 describes the architecture of the hypercomputer platform designed in our group, nicknamed THE PARFORM. With a new and optimized design, aiming primarily at high performance for special applications, we achieve speedups which are higher than in similar approaches. They even are very close to those of a tightly coupled multiprocessor system. We developed new dynamic load balancing mechanisms which guarantee high performance even in the case of dynamically varying workstation loads. Only this approach follows the idea of utilizing idle resources to the end. Section 3 describes the parallelization of an explicit heat equation solver with THE PARFORM. Section 4 deals with the application interface. In section 5 we present our practical experiences with THE PARFORM and similar platforms. We conclude with an analysis of scalability and provide an analytic model. Finally, we identify further problems, whose solution is important for a successful future of the hypercomputer idea.

The following two fundamental issues arise in hypercomputer environments: For a programming style, reasonably independent from network topologies and specific workstation addresses, a mechanism for providing *location transparency* is necessary. In heterogeneous environments the *data representations* of different architectures should be converted without programmer interaction. These two issues are well known in distributed systems [4].

At the present state of the art it seems virtually impossible to get high speedup by applying automatic parallelization techniques to arbitrary programs. This is even true for classical static, homogeneous, tightly coupled architectures. Therefore, a suitable *parallel programming model* has to be provided to the programmer. Our comparison with existing approaches will show, that synchronization and communication concepts play a central role for the performance of such platforms.

Furthermore, the hypercomputer has to decide, which machines will be used and how large the individual subtasks for an efficient parallel computation should be. These decisions call for *static partitioning* and *scheduling heuristics*. Moreover, the load situation in a hypercomputer environment varies frequently, caused by short to medium jobs or by longer production runs. To avoid noticeable interference with interactive programs, a *local priority regulating mechanism* is necessary. In applications with tight restrictions on subtask synchronization, even small changes in workstation loads can significantly slow down the entire computation. Hence, *dynamic load balancing mechanisms* must be used to reschedule the load in such situations. This may be done by dynamically adapting the size of the subtasks or, in extreme cases, by migrating subtasks.

2 The Architecture of THE PARFORM

THE PARFORM consists of three kinds of processes: The administrative process, several executive processes and load sensors. They fulfill all the tasks to provide a network transparent interface between THE PARFORM, the programmer and the application. Furthermore, all supported load balancing strategies base on these services. The processes and their functionalities are described in the following subsections.

2.1 System Startup

At system startup the *administrative process* determines the number of workstations running in the network and their load situation. As broadcasting techniques tend to clog the network and may pose problems when crossing gateways, the detection of idle workstations and the collection of load data is based on a prearranged file of Internet addresses. Our protocols use connectionless communication and asynchronous interrupt techniques, and have a built-in fault tolerance mechanism. This technique yields total startup times of THE PARFORM of about one second for some forty machines in a LAN and is thus faster and more reliable than similar systems like PVM for example. From the information of the load sensors the administrative process determines those workstations, which will be used in the parallel computation.

Based on information given by the user and the previously determined total number of workstations, the administrative process computes the communication graph for the subtasks. Next, the administrative process places one executive process on every employed workstation and starts a protocol for setting up communication links between these subtask processes. This is done by mapping communication links to Internet socket connections. All necessary protocol steps are hidden from the application which views the communication links as UNIX file descriptors to be accessed by read-write-like system calls of THE PARFORM. When the communication graph has been completed, the application dependent part of the administrative process sends the application data and partitioning information defining the specific subtasks to the remote processes. During the parallel computation the administrative process may be used for monitoring workstation loads and the progress of the computation. Finally, it collects the results for storage or further processing.

An *executive process* consists of application independent code for setting up communication links, as well as application dependent code for receiving input and output data from the administrative process, and for performing arbitrary sized subtasks of the total computation. The executive processes are started by the administrative process via the Internet daemon. This technique guarantees short startup times.

2.2 Task Generation

The task generation component of the administrative process performs the *quantitative parallelization*, optimizing the number and size of the parallel subtasks with respect to the actual load situation.

There are several existing approaches for automatic self-scheduling of tasks. In the well known LINDA approach [1],[6],[11] this is done using a distributed tuple space. Worker processes scan the tuple space for tasks to be performed. Whenever such a task is found, it is retracted from tuple space, performed, and the result is written back into tuple space. Subtask communication also is realized by accessing the tuple space using suitable tuple formats. One main advantage of this approach is its flexibility, allowing the same program to be run on different architectures and numbers of workstations. The other is its simple and straight forward paradigm for parallel programming. The interface to the distributed tuple space offers architecture independence, the mechanism by which the tuple space is scanned ensures self-scheduling of the tasks. However, the programmer usually determines the number and size of the tuples before compile time. Therefore, both are independent from the load situation, architecture, and size of the network. This poses problems, if a task is divided into an inappropriate number of subtasks. The main bottleneck of LINDA in a distributed environment is the concept of tuple space, especially the necessary scanning operations to find tuples of certain formats [11].

In our approach though, size and number of the subtasks are optimized with respect to the present load situation analyzed by the load sensors of the platform. A heuristic, adapted to the problem, determines division-parameters, which characterize the subtasks. This calls for some additional programming, since parts of the task generator and scheduler are application dependent, but yields much better speedup.

2.3 Load Balancing

We distinguish the following load balancing strategies:

Static and homogeneous partitioning divides the task into equally sized subtasks, not taking into account the different load situations and capabilities of the workstations.

Static and heterogeneous partitioning divides the task according to the actual load value measured at the individual workstations. As a result, workstations able to contribute more to THE PARFORM automatically are assigned larger subtasks, guaranteeing optimal load balancing at startup time. Consequently, less computing power is wasted in idling CPUs and the total elapsed time is shortened. However the measured load values only reflect the load situation of the workstations at the time of the measurement. Chances are, that sooner or later this load situation changes dramatically and that we get stuck with an inefficient partitioning of subtasks.

We solve this problem with *dynamic load balancing.* At certain times during the computation the workstations exchange their load values and adapt the size of the subtasks to the actual load situation. Suppose a user starts a long production run on a workstation presently contributing to a parallel computation. Consequently, the load of this workstation increases and THE PARFORM determines, that this workstation should shrink the size of its subtask. Therefore, parts of its subtask are transmitted to workstations to which a communication link has been established already for intertask communication. Before actually changing the sizes of the subtasks an inexpensive protocol must ensure, that this communication partner in fact is ready to receive additional chunks of work with respect to its own resource and load situation. This protocol is added to the normal subtask communication in a huckepack fashion to avoid additional message transfer calls.

In a hypercomputer environment there are several aspects, which make dynamic load balancing an indispensible technique to improve performance and scalability. It is possible that the initial assignment of work to the individual machines does not fully utilize their processors. We usually do not have an exact model telling us how much load a subtask of a certain size actually will place on a workstation, since this depends on details of the specific application, the architecture of the workstation and other, partially unforeseeable phenomena. Due to system activities, other than the parallel computation, the load situation of the individual workstations and the connecting network varies during the entire computation. Especially for problems with tight requirements on subtask communication, the delayed execution of a single subtask increases the answer time of the computation as a whole.

Therefore, it is necessary to design mechanisms to change the size of a subtask during the computation. Such techniques can be involved for certain classes of applications but may be too complex to be designed for arbitrary situations. However, their design is straight forward for problems with static data flow graphs, using the functionalities of the platform. The necessary software engineering effort will therefore quickly pay off for production runs of large-scale problems.

Different dynamic load balancing mechanisms which could easily be designed with the help of our platform also include subtask migration. For lengthy tasks in insecure environments it is also possible to take snapshots of subtasks. However, corresponding functionalities for THE PARFORM are still under development.

2.4 Load Sensors

For parallelization and scheduling we need to know, which machines of a network are running at all, and which performance THE PARFORM could expect from which machine. For this purpose we utilize two kinds of data:

Idle time percentage is obtained by periodically reading UNIX kernel variables. *Available performance* information is calculated by interrupting a loop producing a small dummy load after certain periods of time and evaluating the number of loop executions afterwards. The load data are collected by the administrative process via an asynchronous connectionless socket interface. Multiplexed communication ensures low overhead, retry ensures a response in case of packet losses.

Workstations running at most interactive or small jobs may be identified by idle time percentages with sampling frames from several seconds to several minutes. For these workstations the second type of data, the available performance, informs us about the possible contribution of this workstation to the overall computation. Scheduling appropriate parts of our job to those machines, we exploit time slices during which the workstation otherwise would be idle.

Workstations with a very small idle time percentage run computationally intensive jobs. Here, the available performance value tells us, how much computing power we would obtain from this workstation, if we participated in the CPU-sharing. However, scheduling parts of our job to such machines would also prolong the answer time for all the other jobs. Using both data, we are able to construct a reasonable scheduling heuristic.

As permanent dynamic changes of the load situation are characteristic for a workstation environment we can tolerate mildly out-of-date information. Also the number of active workstations changes from time to time. Thus, the performance reporting service is built upon an unreliable protocol. However, it is important to access that information instantaneously, without waiting for the startup and availability of load values of the sensors. On the other hand the load measuring tool should not produce too much load. These contrary requirements determine the present implementation where the load sensors are running as background processes on all contributing workstations.

For the dynamic task adaption mechanism, the implemented load sensors also provide actual load information on demand. Furthermore it is possible to obtain information about other system activities like paging, swapping, context switches, Ethernet collisions and the like. A seperate monitoring tool allows us to display those values periodically on the administrating workstation. This immediately gives us a picture of the present load situation of the entire network, bottlenecks may be detected and the overall performance of the scheduling heuristics can be observed.

2.5 Implementation Details

The current implementation of THE PARFORM is written in C on Sun SPARC-stations running SunOS within an Ethernet LAN. We also successfully employed a MicroVax running Ultrix and Silicon Graphics workstations during early phases of our work. The system is implemented on top of the UNIX operating system and the transport layer of the Internet protocol suite, using TCP, UDP and the Berkeley socket mechanisms [9]. Application programs must be written in message passing style assuming distributed memory. Communication preserves the ordering of messages and uses the reliable, bidirectional, connection oriented TCP protocol. Synchronization can be done by blocking platform function calls, which wait until the receipt of a message. Data representation, marshalling, and similar architectural aspects are hidden from the programmer.

We presently do not base our platform on existing communication software, but use optimized approaches of our own. In other systems, numerous protocols and software layers are implemented, which solve the communication problems in a flexible way. But many of them produce much overhead. USENET groups cited "ISO-OSI will tomorrow handle the loads of yesterday whereas TCP today is handling the loads of today". NASA designs the officially required standardized protocols of their space station Freedom around high speed protocols of their own [5]. Nevertheless, porting THE PARFORM to the ANSA network standard [4] is planned for a later phase of the project to study aspects of interfacing with existing distributed system platforms. However, at the cost of additional overhead. More attractive are ports to the MACH operating system [10], since they allow THE PARFORM to use the efficient scheduling and communication primitives of modern microkernel architectures.

3 An Application: Heat Conduction

During the various phases of the development of THE PARFORM a number of examples has been studied, among them the calculation of fractals, cubic convolution algorithms for performing image transforms and partial differential equations. In this chapter we present an explicit solver of a parabolic partial differential equation from the theory of heat conduction. Although a rather trivial example from the numerical and physical point of view, this example has a number of advantages for studying the performance of THE PARFORM: The amount of computation and communication is known in advance. There is a high amount of communication throughout the calculation, and the requirements on a precise synchronization of the individual subtasks are very strict. A delay of a single task immediately slows down the entire computation.

Consider the two dimensional heat equation:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}.$$

This equation describes the temperature distribution $u(x, y, t)$ on a rectangular plate as a function of the coordinates x and y and the time t. The temperature distribution depends on an initial condition on the entire spacial domain at the initial time $t = t_0$ and boundary conditions along the edges known for all times $t > t_0$. Applying an explicit forward difference approximation to the equation we obtain

$$\frac{u_{i,j}^{(k+1)} - u_{i,j}^{(k)}}{\Delta t} = \frac{u_{i+1,j}^{(k)} - 2u_{i,j}^{(k)} + u_{i-1,j}^{(k)}}{(\Delta x)^2} + \frac{u_{i,j+1}^{(k)} - 2u_{i,j}^{(k)} + u_{i,j-1}^{(k)}}{(\Delta y)^2}$$

$$u_{i,j}^{(k+1)} = r_x u_{i-1,j}^{(k)} + r_x u_{i+1,j}^{(k)} + (1 - 2r_x - 2r_y) u_{i,j}^{(k)} + r_y u_{i,j-1}^{(k)} + r_y u_{i,j+1}^{(k)}$$

where $r_x = \Delta t/(\Delta x)^2$, $r_y = \Delta t/(\Delta y)^2$ and $u_{i,j}^{(k)}$ denotes the value of the temperature at grid node $P_{i,j}$ and time $t_k = t_0 + k * \Delta t$. For more details see [8].

Given the initial condition $u_{i,j}^{(0)}$, the values $u_{i,j}^{(k+1)}$ are calculated with the recursive equation $u^{(k+1)} = f(u^{(k)})$. The value of $u_{i,j}^{(k+1)}$ depends on the values of the same grid point and its four neighbours at the previous time step k. This numerical aspect and the equivalent locality of the physical law immediately suggest an idea for parallelizing this problem: We partition the grid of the rectangular domain. For simplicity of implementation we chose rectangular stripes consisting of a set of neighboring columns of grid points and schedule the calculation of temperature u on each set of columns to one workstation. As can be seen from Figure 1, we thereby introduce artificial, numerical boundary conditions between the neighboring tasks. The respective values of u must be communicated between neighboring tasks after each time step, thus imposing tight synchronization requirements on the subtask communication.

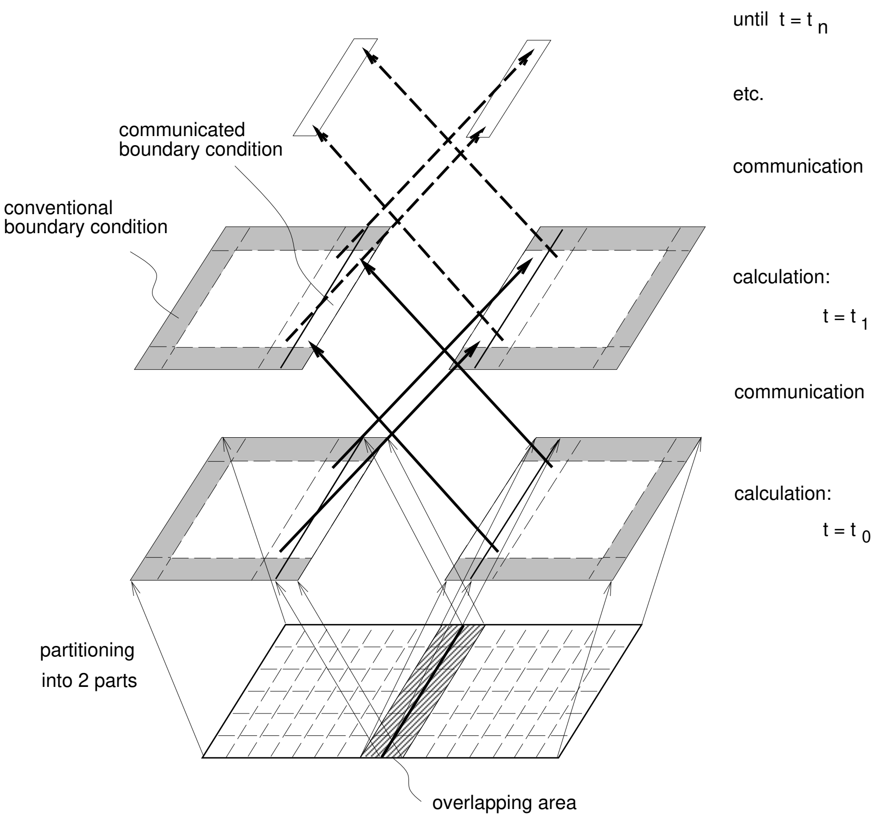

Fig. 1. Grid partitioning and time integration

The *qualitative* parallelization determines the actual implementation of the subtasks based on the decision to partition the grid and thereby establishing the corresponding communication scheme. For efficient load balancing, we additionally have to deal with *quantitative* parallelization by deciding on the the size of the individual subtasks, which is determined by the number of columns of nodes of the grid. The amount of work to be done within a single subtask is approximately proportional to this size. Hence, static partitioning may easily be implemented. Dynamic load balancing may be implemented by transfering a subset of the set of columns to one of the neighboring subtasks.

Static homogeneous and heterogeneous partitioning strategies are implemented by suitably choosing the grid dimensions of the subtasks. Due to the synchronization of the subtasks after each time step the elapsed time for the entire task is determined by the workstation with the least performance. If we use the homogeneous strategy, high performance workstations cannot contribute all of their CPU power but remain idle, waiting for the slowest machine to complete its time step. The heterogeneous strategy partitions the grid according to a load value measured at the individual workstations. As a result, resource utilization increases and the total elapsed time is shortened.

4 The Application Interface

The application is connected to THE PARFORM by macros, library functions and object modules to be linked to the application program. The available functions allow a fully location transparent style of programming. Subtasks are identified by system generated task handles, automatically scheduled to remote sites. Communication is done by accessing communication handles, automatically mapped to UNIX file handles, which are connected to Internet sockets. The library functions supporting communication also perform presentation layer tasks like converting different data representations. Upon startup, flags for the necessary conversions are set as encoded in the host file. By choosing appropriate values for the conversion flags, the user has the choice between sender-makes-it-right, receiver-makes-it-right or XDR-style encoding [3].

Achieving high speedups with the heat equation solver, as described in section 5, requires some parallel programming skills of the programmer, although THE PARFORM presently takes care of many details: Coding the qualitative aspects of the parallelization, especially the parallel code for the subtasks, and providing information about the communication structure between individual subtasks. Quantitative aspects of the parallelization, namely the sizes of the subtasks, are determined by the platform.

So far, THE PARFORM approach is suited for problems with static and regular data flow graphs, like linear algebra problems, partial differential equation (PDE) solvers, Fourier transforms and image processing algorithms. It can be applied to geometrical and combinatorial algorithms, too, but here it might be difficult, to obtain a well balanced load distribution.

All library functions can be used in various debug modes to enable tracing of every detail during remote program startup, establishment of the socket connections and platform activities during the calculation. Full communication snooping on the links is possible and proved to be an excellent debugging aid, when porting a sequential program to THE PARFORM.

Parallelizing an existing sequential program for THE PARFORM consists of the following tasks: Restructuring the algorithm for parallel computation, adding communication primitives to the computational kernel and minor restructuring of the code. Communication stubs must be designed for transmitting application data between the administrative process and the executive processes, code must be written for defining the communication topology for an arbitrary number of tasks. Next, we have the choice between static homogeneous or heterogeneous partitioning, and have to decide on dynamic load balancing. To complete the adaptation, several small modifications like the inclusion of header files must be done. We will try to reduce the additional implementation work using libraries of partitioning routines, which easily may be built for classes of regular problems, like PDE solvers, signal transforms and the like.

5 Practical Experiences

In this section we present a number of experiences with the implementation of the heat equation. All network distributed computations were performed in the LAN of our department consisting of 23 SPARCstation1, 8 SPARCstation1+, 4 SPARC-station2, 2 SPARCserver490 and 1 SPARCserver390. For comparison with a tightly coupled multiprocessor we carried out computations on our Transputer Multicluster MC-2/32-2, programmed in OCCAM. Runtimes of the measurements on the dedicated local network are listed in Table 1 below.

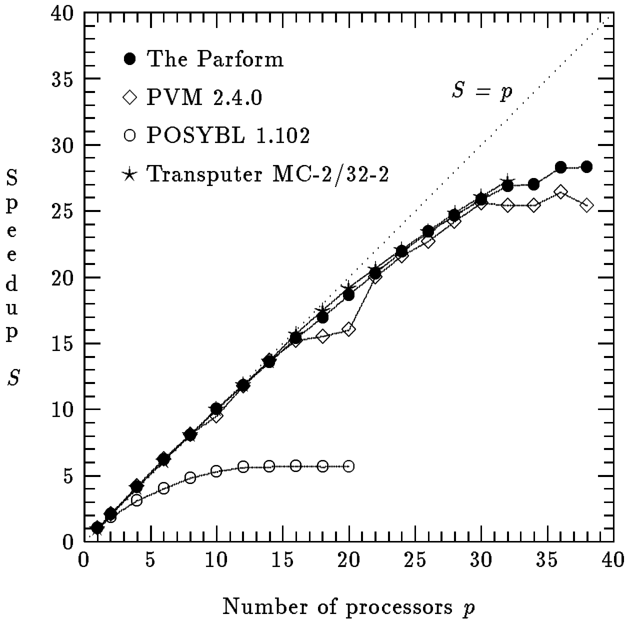

Fig. 2. Heat solver with static, homogeneous partitioning.

Figure 2 shows the speedups for the heat equation with static, homogeneous partitioning, computing 500 time steps on a 3800 $\times$ 100 grid using THE PARFORM, LINDA (POSYBL 1.102) and PVM (version 2.4.0 with the new, fast communication primitives) on the network, and the OCCAM program on the Transputer. Since the slowest task of the parallel computation determines the overall speedup, Figure 2 shows the speedup with respect to the sequential run on the slowest workstation in our LAN, a SPARCstation1.

The poor performance of the LINDA implementation POSYBL can thoroughly be explained by the overhead of tuple space management. With 10 to 20 processors the speedup essentially remains constant. Although work on high speed tuple space implementations is in progress [11], tuple space management is a principle bottleneck of the LINDA approach, rather suited for shared memory architectures than for distributed systems.

Processors	Linda	PVM	The Parform, homogen. Part.	The Parform, heterogen. Part.
1	1370.6[1]	1370.6[1]	1370.6[1]	564.0[2]
2	737.2	648.0	654.8	336.9
4	442.6	328.0	332.3	173.9
6	339.3	219.0	221.7	122.8
8	284.6	168.4	170.2	106.2
10	260.2	143.6	137.4	92.6
12	244.7	116.6	116.0	80.1
14	242.7	100.1	103.5	73.5
16	239.5	90.0	89.0	65.3
18	242.6	97.5	80.9	61.9
20	241.6	85.8	73.5	60.4
22		68.5	67.5	56.2
24		63.6	62.5	52.3
26		60.5	58.6	51.6
28		56.7	55.8	49.6
30		53.5	53.0	48.5
32		54.0	51.0	47.4
34		54.0	50.8	45.2
36		52.0	48.5	45.3
38		54.0	48.4	48.6

[1] SPARCstation1, [2] SPARCstation2

Table 1. Runtimes for heat solver in seconds.

The Parallel Virtual Machine PVM, developed by Dongarra and others [2] is a very similar approach to The Parform. Measurements with PVM version 2.4.0 using the new fast communication primitives *vrcv* and *vsnd* show lower performance than The Parform. Several concepts in PVM, for example the startup phase or the remote daemons produce more overhead than The Parform. Load balancing is not supported in PVM. It has to be done by the programmer himself and therefore high efficiency is hard to achieve, especially if the network runs under normal daily load.

The Transputer Multicluster is slower than THE PARFORM, approximately by a factor of three. The reason is the lower performance of the T800 processors compared to the Sparc processors. The most interesting observation in Figure 2 is that the workstation network scales exactly like a tightly coupled multiprocessor when solving the heat equation.

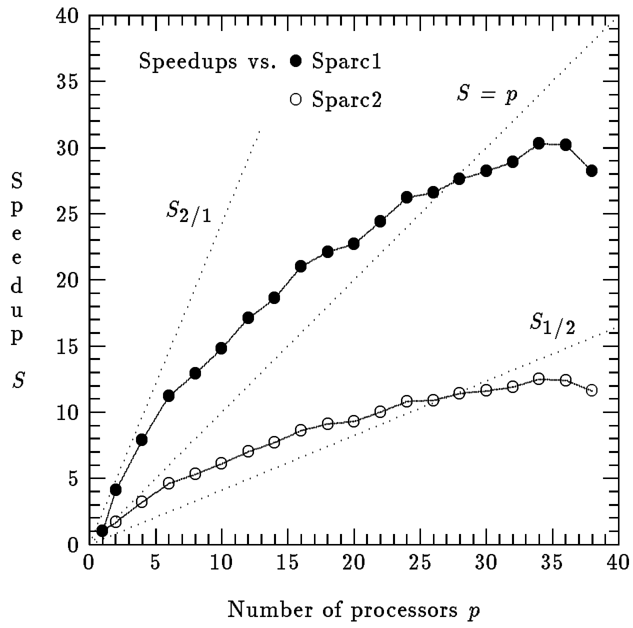

Fig. 3. Heat solver with static, heterogeneous partitioning.

Figure 3 shows speedups using a static, heterogeneous strategy, with the subtask sizes automatically adapted to the individual machine capabilities. While the static, homogeneous approach yields equally sized partitions, the heterogeneous produces task sizes proportional to the machine performance. In the parallel computations with p processors we employed the p fastest machines of our LAN.

In case of homogeneous partitioning the slowest machine in the net is responsible for the speedup (see Figure 2). All faster machines idle while waiting at the synchronization points, which are the communication statements. Thus, it is reasonable to calculate speedup values with respect to the runtime on the slowest machine. With heterogeneous partitioning we cannot use the conventional definition of speedup, because of the different workstation performances. We could introduce a scaled speedup with respect to an average machine performance, but the dotted lines in Figure 3 offer more information for assessing the quality of the speedup. The diagonal represents the ideal speedup $S = p$. The other two dotted lines limit the speedup with respect to the different workstation performances. The upper line shows the ideal, linear speedup $S_{2/1}$ of p SPARCstation2 with respect to the sequential SPARCstation1 version, the lower dotted line the ideal speedup $S_{1/2}$ of p SPARCstation1 with respect to the sequential SPARCstation2 version. The speedup measured in a net consisting of different types of machines is limited by the dotted diagonal and the lower or upper dotted lines, respectively. The speedup curves in Figure 3 are bent above the limit due to superlinear speedup caused by swapping and below the limit because of scalability degradation. Network congestion, a special effect of scalability degradation occured with 34 processors or more. Identical experiments produced varying runtimes with differences in the order of several seconds.

In order to analyze dynamic load balancing we solved the heat equation on a 3800×100 grid for 2500 time steps on 20 machines (SPARCstation1 and SPARCstation2). Due to the presence of the faster SPARCstation2 machines the heterogeneous partioning will be more efficient than the homogeneous. Neither homogeneous nor heterogeneous partitioning can react on load changes after system startup. We compared the different load balancing strategies on a dedicated network with all machines idling except one SPARCstation1 where an infinite loop was started after THE PARFORM startup. When starting the loop the local UNIX scheduler shares the CPU equally between the loop and the task of the parallel computation. Therefore, we would expect the statically partitioned program to slow down by a factor of 2. Our dynamic load balancing strategy almost completely distributes the work of this task to the neighboring tasks. Only a minimum amount of work stays at the processor running the loop. Thus, we regain utilization of all other CPUs. THE PARFORM produced the following execution times:

Static, homogeneous partitioning	$742.0s$
Static, heterogeneous partitioning	$536.0s$
Dynamic load balancing	$354.0s$

These results for dynamic load balancing are obtained with a very simple protocol. Various effects arise from the dynamics of transfering chunks of work between the tasks. Interesting analogies with classical wave theory seem to appear in the resulting load distribution.

If the necessary field length of an application surpasses the amount of physical memory one often observes excessive swapping and paging activities. In the case of local physical laws discretized on a grid, numerical methods access the individual grid points at rather high frequencies, leading to a permanent paging on the grid. Since important problems like fluid dynamics or weather simulation often have grids with some thousand points in all three dimensions, such problems usually cannot be solved in workstation environments with typically some 32 Mbytes of physical memory. In our measurements we experienced superlinear speedups of two kinds.

For a grid size of 7800×200 we found extremely high speedup values, because the sequential version was paging more than 10 hours, before we aborted it. The same problem with 38 machines resulted in a runtime of 154.0 seconds. Here, the physical memory of the workstations was large enough for the computation. If programs require so much memory, that some other processes have to be swapped out before this program can be executed, this leads to slightly superlinear speedup.

We briefly want to mention some experiences with user reactions. There are psychological obstacles to "stealing" performance from the "owner" of a workstation. Feedback of the department staff during our experiments was nevertheless encouraging, since often our activities went unnoticed by the users. From time to time we were suspected of effects, which turned out to come from a totally different source. Our parallel computations sometimes interfered with other production runs. We therefore plan to design a priority system, which assigns lower priorities to production runs and programs like screenlocks but higher priorities to interactive programs.

6 Scalability

In Figure 2 we observed an excellent, nearly linear speedup when using up to 20 workstations. However, the speedup breaks down for more than 30 workstations. There are four possible reasons: *Saturation*, *desynchronization*, *protocol overhead* and *network congestion*.

Saturation means, that a problem is split into so many subtasks, that the parallelization overhead, mainly communication, outweighs the performance gained by the parallel computation. This phenomenon can only be influenced by choosing different algorithms or parallelization strategies.

Desynchronization occurs, if workstations have to wait for data overdue from other machines. This may happen, if sudden activities on the network cause a specific machine to reduce the CPU share of the process which is part of the parallel computation. This is a general problem for applications with strict constraints on synchronization, like the heat solver. All machines must wait before calculating the next time slice, if only a single machine is late.

Even the best network technologies are not able to avoid *protocol overhead*, that part of the communication, which is needed for setting up drivers and adjusting flow control parameters.

Network congestion occurs, when the rate of communication comes close to the throughput of the network. This phenomenon is a physical limitation of the communication medium. As it is well known [7], this is more likely to be a problem with CSMA/CD type networks like the Ethernet than with token-passing techniques. With the random-access technique of the Ethernet collisions might occur when many stations try to transmit a frame at approximately the same time. This is the case during the communication periods of the heat solver and leads to reduced speedup. Communication technologies of the next years, offering up to 10 Gbit per second bandwidth might solve this problem. In the mean time it may partially be handled by intelligent communication patterns.

In order to get a better understanding of those effects we developed an analytic model. We suspect that the sublinear speedup behavior (Figure 2) up to about 34 processors is mostly caused by saturation and the breakdown with more than 34 processors by Ethernet congestion. To show this, we model the speedup of the heat solver including the collision behavior of the Ethernet.

Solving the heat equation on an $m \times n$ rectangular grid for T time steps yields a sequential computation time of $t_s = CmnT$. The constant C denotes the average time for evaluating a single grid point. On a SPARCstation1 we found $C = 7.21\mu s$. To calculate the elapsed time of the parallel computation, we also have to integrate the total communication time Tt_{com} into our model, where t_{com} is the average communication time per time step. Let the axis of the rectangular domain, which is discretized into m grid points, be partitioned into p parts for a parallel computation on p processors. From our experiments (Table 1) with two processors we can reasonably assume, that the calculation phase is completely parallelizable. Thus, the elapsed time of the parallel execution can be calculated as

$$t_p = \frac{t_s}{p} + Tt_{com} = \frac{CmnT}{p} + Tt_{com}.$$

During every time step each processor transmits $2n$ double numbers (8 bytes each), n to the left and n to the right neighbor, except the processors computing the boundary tasks. Hence, during one communication phase $2p - 2$ frames are transfered. A frame comprises n double numbers plus header and trailer information, which consists of 18 bytes in the Ethernet standard. In our case with $n = 100$, the frame contains 818 bytes. The frame length f is the quotient of the number of bits of the frame and the cable transmission rate, which is 10 Mbit per second in a standard Ethernet. Thus, $f = 0.65ms$ for our 818 byte frames.

Ethernet modeling in [7] offers the following formula for the normalized transfer time γ of a single frame, assuming constant frame length f:

$$\begin{aligned}\gamma &= \rho\frac{1 + (4e+2)a + 5a^2 + 4e(2e-1)a^2}{2[1-\rho(1+(2e+1)a)]} + 1 + 2ea + \\ &\quad + \frac{a}{2} - \frac{(1-e^{-2\rho a})(\frac{2}{\rho} + 2ae^{-1} - 6a)}{2(e^{-\rho(a+1)-1} - 1 + e^{-2\rho a})}.\end{aligned}$$

e is the base of the natural logarithm. $a = \tau/f$ depends on the architecture of the Ethernet, where τ denotes the end-to-end propagation delay, which we estimated to $7.8\mu s$ in our LAN. $\rho = \lambda f$ denotes the traffic intensity, where λ is the total average traffic in frames per second. Our load monitor suggests the approximation that packet submission is equidistributed during the average communication time t_{com}, because mutual exclusive access to the Ethernet causes desynchronization of the tasks. This will be valid for communication phases with a moderate number of collisions. However, for traffic with heavy collisions, the assumption will merely produce an optimistic speedup approximation. During one communication phase, $2p - 2$ frames are transfered. Therefore, we obtain $\lambda = 2(p-1)/t_{com}$ and for the traffic intensity

$$\rho(t_{com}) = \frac{2(p-1)f}{t_{com}}. \tag{1}$$

Given frames of constant length f (in units of time), $f\gamma$ is the average transfer time of one frame. We therefore get an equation for t_{com} expressing the highly nonlinear behavior of the network

$$t_{com} = 2(p-1)f\gamma(\rho). \tag{2}$$

Substituting (1) in (2) we obtain a nonlinear equation in ρ which is solved iteratively. The resulting speedup values are shown as the solid lines in Figure 4.

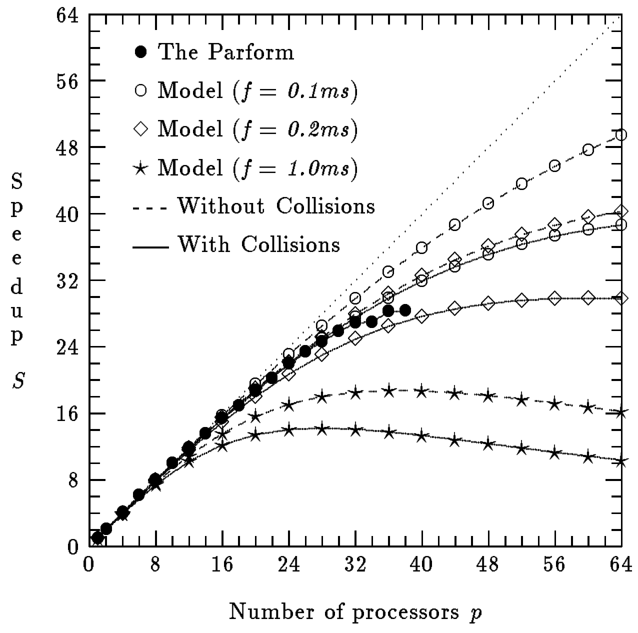

Fig. 4. Scalability model of our heat solver.

We chose the frame length f as the only fitting parameter of our model. With $f = 0.1ms$ our model fits the measured curve, while the others show the speedup for varying frame length f.

We now can explain the sublinear speedups with the help of this model. Setting $\gamma = 1$ gives the speedup without considering collision effects, leading to the broken lines in Figure 4. We get a qualitative notion of the saturation effect from Figure 4 if we interpret the frame length f as the ratio of the communication and calculation parts of the computation. Thus, the curves with larger f correspond to computations on smaller grid sizes. Including Ethernet specific effects further flattens the speedup curves (solid lines). Our computation with THE PARFORM just enters the region where collisions become noticeable when calculating with 34 to 38 processors. In principle the Ethernet performance suffers from collisions when the traffic intensity ρ increases. The average transfer delay increases in an unbounded way if ρ approximates its maximum value $\rho_{max} = 1/(1 + (2e + 1)a)$ (see first term of the formula for γ). This leads to increasing values of t_{com} and a decreasing speedup curve. Thus, the analytic model gives a fairly good prediction of the experienced speedup and its break down, which is due to saturation and network congestion.

7 Conclusions and Further Work

The presented results and experiences with our platform show that we can regard a workstation network as a tightly coupled multiprocessor system, when exploiting its resources with a system like THE PARFORM. Our dynamic load balancing strategy yields speedups superior to conventionally used static partitioning. This static approach does not work well in a workstation environment with its changing load situation. First performance evaluations indicated very low overhead, which is due to the compact and new design of THE PARFORM. Actual response times, however, depend on the load situation of the network.

The scheduler of the hypercomputer interacts so well with the operating system, that the average user does not notice any other activities on his workstation in addition to his own. On the other hand, a workstation which only performes local editing or email functions easily contributes around 90% of its computing power to the hypercomputer.

The platform currently works fine for PDE solvers. There are many different parameters and heuristics whose effects on overhead and performance still have to be evaluated. Further performance measurements and system fine tuning under varying and well controlled artificial load situations are necessary. We intend to implement a number of different application problems to get further ideas, how the application interface can be improved. Furthermore, we have to gain experience about the advanced reconfiguration mechanisms of the task monitor and want to study how the system scales to configurations with several LANs connected by gateways.

Presently, the platform aims at users familiar with parallel programming. It offers language extensions for parallel, location transparent programming in a heterogeneous, distributed environment. For applications with a static data flow graph, essentially known at compile time or after a number of input parameters are given, interesting perspectives for a user interface are possible. In situations, where the data flow graph dynamically evolves with the computation, as this is usually the case with recursively defined functions and algorithms on data structures like trees, this might not be possible.

Finally, we want to emphasize that the platform approach will not only be useful in distributed workstation networks or networks with different kinds of machines. We believe that also massively parallel computers will benefit from concepts like The Parform.

8 Acknowledgments

We want to thank our system specialists Beat Rageth and Rico Solca for continuous support, we hope to benefit from further interesting discussions with Edgar Lederer and thank Prof. Lutz Richter for his supervision. The research staff of our department provided support by giving us the chance to run experiments with all available workstations. We even could shut down the entire network for some late night measurements under controlled network situations.

References

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

[2] A. Beguelin, J.J. Dongarra, G.A. Geist, R. Mancheck, V.S. Sunderam, A User's Guide to PVM Parallel Virtual Machine, Oak Ridge National Laboratory, ORNL/TM-11828, July 1991.

[3] B. Lyon, *Sun External Data Representation Specification* (Sun Microsystems, Inc., Mountain View, 1984).

[4] S. Mullender, *Distributed Systems* (Addison-Wesley, New York, 1989).

[5] A.C. Weaver, XTP for the NASA Space Station, *Protocols for High∼Speed Networks*, H. Rudin, R. Williamson, ed. (North-Holland, Amsterdam, 1989).

[6] G. Schoinas, Issues on the Implementation of Programming System for Distributed Applications, Draft Paper, University of Crete (1991).

[7] M. Schwartz, *Telecommunication Networks: Protocols, Modeling and Analysis* (Addison-Wesley, Reading, 1987).

[**8**] G.D. Smith, *Numerical Solution of Partial Differential Equations: Finite Difference Methods* (Oxford University Press, New York, 1985).

[**9**] W.R. Stevens, *UNIX Network Programming* (Prentice-Hall, Englewood Cliffs, 1990).

[**10**] A.S. Tanenbaum, *Modern Operating Systems* (Prentice-Hall, Englewood Cliffs, 1992).

[**11**] G. Wilson, ed., Linda-Like Systems and Their Implementation, Technical Report 91-13, Edinburgh Parallel Computing Centre (June 1991).