A Monitoring System for a Transputer-Based Multiprocessor

Mats Aspnäs Thomas Långbacka

Åbo Akademi, Department of Computer Science
Lemminkäinengatan 14, SF-20520 Åbo, Finland

October 1, 1990

Abstract

This paper describes a distributed performance monitoring system for a transputer-based multiprocessor, Hathi-2. The monitoring system allows the programmer to measure how well a parallel program utilizes the hardware resources of the multiprocessor system, i.e., the processor CPU's and communication links, during program execution. It can be used for identifying performance bottlenecks in parallel programs. We present the different components of the system, together with a case study where we monitor an existing application program.

1 Introduction

When writing parallel programs, one tries to acheive a speedup that is proportional to the number of processors, i.e. linear speedup. Often, however, given a fixed size problem, the measured speedup is lower than expected and approaches asymptotically some fixed limit when the number of processors are increased. This indicates that the parallel program contains some bottlenecks that make it impossible to utilize all processors efficiently.

A bottleneck in a computer system is generally caused by some resource that is not able to process data at a rate that matches the rest of the system. As a concequence, other resources in the system will become idle and the maximum efficiency of the system can not be fully utilized. In a distributed MIMD multiprocessor system, like in a transputer-based machine such as ours, bottlenecks are ususally casused either by overloaded processor CPUs or communication links.

Without any assisting tools, performance bottlenecks in distributed programs are difficult to identify. Generally, one can not analyze the dynamic behaviour of a parallel program just by examining the source code. Thus, if the program executes slower than expected the reason for this is often difficult to find. To be able to locate performance bottlenecks, one must be able to monitor the execution of the program.

When the programmer has identified a performance bottleneck in the program, he must analyze the program and find out what caused the bottleneck and try to eliminate it. In some cases bottlenecks can easily be removed by changing the algorithm and/or the configuration of the processor network on which the algorithm is executed. In some cases it might be very difficult or impossible to eliminate a bottleneck. Not all problems can be solved efficiently on a multiprocessor system.

In this paper we present a system for monitoring the transputer-based multiprocessor system Hathi-2. Our main interest is performance monitoring, in contrast to diagnostic monitoring, which is more related to (high-level) debugging. Below we list a number of problems that we have concerned ourselves with in monitoring performance of distributed systems.

Monitoring often *affects program execution.* In a sequential system, this only means that the program execution is delayed. In distributed systems, a delay on one component can have cumulating effects on other components. This problem is often solved by introducing dedicated hardware for monitoring. This can be very expensive if we are not willing to tolerate any interference on the computation. In that case, the monitoring system can require as much hardware as the monitored target system [4], [15]. Often pure hardware monitors are very inflexible and hard to adapt to changing requirements. Pure software monitors, on the other hand, are always intrusive, but they usually provide flexibility and the possibility to be adapted to individual needs [9]. A hybrid monitoring system, i.e. a system consisting of both hardware and software components, tries to combine the flexibility of software monitors with the efficiency of hardware monitors. For our system, we have chosen the hybrid approach.

Large amounts of data have to be handled by the monitoring system and transported out from the target system. Event-driven systems, in contrast to sampling systems, are especially problematic in this aspect. The data gathering process should not use the same communication paths as the target system, as that will affect the communication behaviour. This is crucial in point-to-point networks, like transputer systems, where the number of communication paths is limited. In our system monitoring data is transported via additional communication paths, designed for this purpose. Similar techniques have been used in other transputer based systems e.g. [8],[12].

In systems built by inserting data collection probes in the operating system, there is often some kind of filtering process that selects (and possibly reduces the size of) data that is of interest to the user (see for instance [11]). In [14] an interesting approach to this activity is introduced. The monitored system is treated like a database, which can be examined using a specially designed database query language. The queries will activate the probes actually needed to produce the answer to the query (the probes will always be present but activated only when necessary).

Even though uninteresting data might be filtered away, there will still be large amounts of monitoring data that have to be *presented to the user in an understandable way.* In traditional sequential systems, various types of diagrams and other types of tabular data have been used. These traditional methods of presentating monitoring data has also been used for more complex systems. For instance, tables or histograms [7], [17] in more or less sophisticated forms have been used. For the programmer to be able to understand what really happens during program execution, the presentation tools have to relate the monitoring data to the logical structure of the program. This kind of approach is used for instance in [4],[6]. In our system we have integrated the monitoring system into a programming environment in which the user draws the logical structure of the program in form of a process graph. The monitoring data is presented upon the same graph.

The rest of this paper is organized as follows: in Sections 2 and 3 we present the hardware and software components of the monitoring system. Section 2 also contains a brief presentation of the Hathi-2 architecture. Section 4 describes how the data generated by the monitoring system is presented to the user. In Section 5 we present a case study in which the monitoring system has been used to identify bottlenecks in a non-trivial parallel program. Finally, Section 6 contains conclusions and some suggestions for future work.

2 The Hathi-2 Multiprocessor System

Hathi-2 [3] is a reconfigurable general purpose loosely coupled MIMD multiprocessor system consisting of 100 32-bit IMS T800 transputers [10]. The system has a distributed switching network built of 25 IMS C004 crossbar switches (one per board) and a similarly distributed

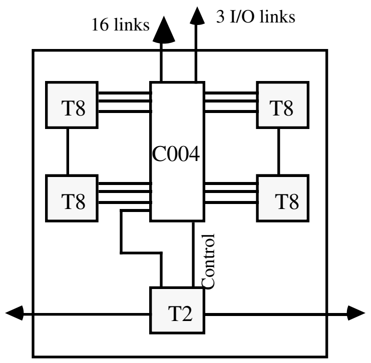

Figure 1: The Hathi-2 Board

control system consisting of 25 IMS T212 16-bit transputers.

The Hathi-2 system was developed in the Hathi project (1986 – 1988) by the Department of Computer Science at Åbo Akademi and the Technical Research Center of Finland (VTT/TKO) in Oulu.

2.1 Architecture

Hathi-2 consists of 25 identical boards. Each board contains four 32-bit T800 transputers, one 32 link C004 crossbar switch and one 16-bit T212 transputer. The T800 transputers on a board are pairwise connected via one link, and the three remaining links are connected to the C004 crossbar switch (see Figure 1).

One of the links on the T2 transputer is used for controlling the setting of the crossbar switch, and another links is connected to the switch. The two remaining links are used to connect all 25 T2 transputers into a ring, thus forming the distributed control system.

Three links from each crossbar switch are reserved for connections with the external world. These I/O links are used for connecting the user's host computer or external devices like disks and graphics controllers to the system. The remaining 16 links from each crossbar switch are used to form a static torus connection between the boards. From the crossbar switch on each board, four links are connected to the left, right, upper and lower neighbor respectively (see Figure 2). Thus, a two-level distributed switching network is formed: a static torus connection between the boards in the system and a reconfigurable level of connection through the C004 crossbar switches [1].

Hathi-2 can be used by several users simultaneously by dividing the system into a number of smaller independent partitions (see Figure 3). Each partition is allocated to one user at a time, and it is connected to the user's host via the I/O links on the first board in the partition. The Reset, Analyze and Error signals are connected from the users host transputer as a pipeline through all transputers in the partition. The user's host is normally a workstation, equipped with a transputer board on which the programming environment is executed. Programs are edited and compiled on the user's host, and the executable program is downloaded onto the multiprocessor system by a network loader.

2.2 The Control System

The T2 transputers are connected into a ring and form a distributed control system (see Figure 4), which is connected to a separate host computer and totally independent of the rest of the

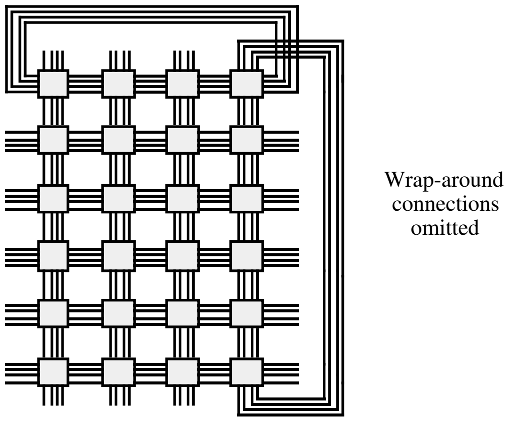

Figure 2: The Inter-Board Connections

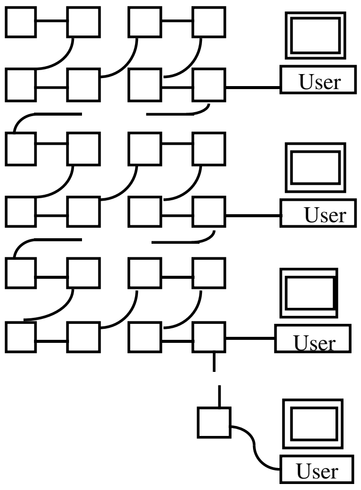

Figure 3: Partitioning the Hathi-2 System

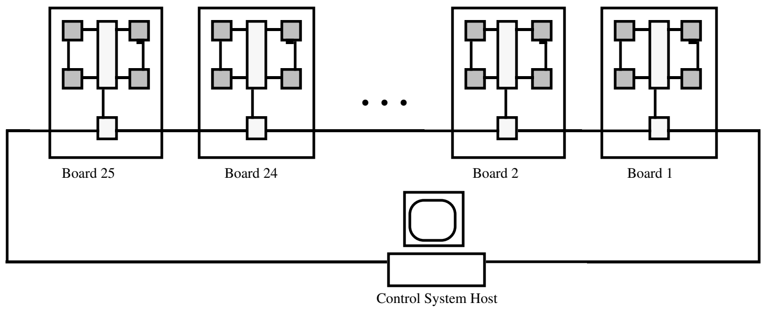

Figure 4: The Control System

system. The only connection between the control system and the user is via an I/O-link, which can be connected to a processor in the control system.

The control system has two main tasks:

- to administrate the *configuration* of the processors in a partition by controlling the settings of the crossbar switches
- to *monitor* the calculating T800 processors in a partition and forward the collected monitoring data to a node where it can be stored in a file, analyzed and presented to the user.

The control system software is based on a simple message-passing kernel, which handles transparent message forwarding between processes in the control system.

2.3 Hardware Support for Monitoring

The Hathi-2 architecture contains two types of hardware devices which support nonintrusive resource monitoring during program execution: a *CPU load meter* and a *link-independent communication path* from each target processor to the control system.

The CPU load meter measures the utilization of a transputer's CPU during a time interval by observing the activity on the data/address bus. It is implemented by a counter which is enabled when there is activity on the bus. The value of the load meter counter register can be read at regular sampling intervals by the processors in the control system. The value gives a direct measure of how much work the CPU has performed during this time. A typical interval length is 50-500 ms, corresponding to a sampling rate of 2-20 per second. The CPU load meter is a passive device which only observes the traffic on the address/data bus. It does not affect the program behavior in any way and needs no additional code to be inserted in the monitored program.

The link-independent communication path connects each calculating T8 transputer with the control transputer on the same board. It is implemented by a 512 byte FIFO buffer mapped to the address space of both processors. A target processor can send a message to the control system by writing a stream of bytes to the FIFO buffer. The control processors read messages from the FIFO buffers and forward them through the control system ring (see Figure 5).

These additional communication paths are used in the monitoring system for sending reports about link communication to the control system. If the reports were sent via the communication links, they would affect the behavior of the monitored program. To use this link-independent communication path, the user has to insert some code into the monitored program, that writes a message to the FIFO buffer. The FIFO buffers can be used for sending any kind of data, i.e., they could be used for implementing high-level debugging and program animation tools.

The Hathi-2 architecture also contains an interrupt subsystem, which is built on the transputer's EVENT interrupt. The control system can generate a hardware interrupt signal which

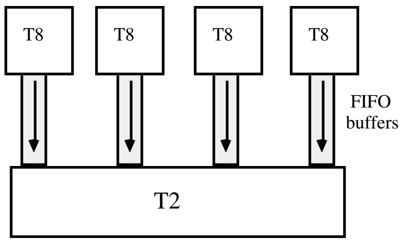

Figure 5: The FIFO buffers

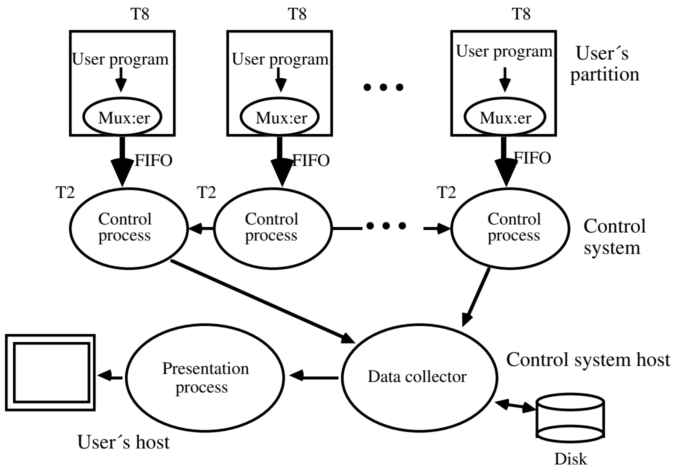

Figure 6: The Processes in the Monitoring System

reaches all processors in a partition simultaneously. This is used in the monitoring system for generating a global polling signal to all target processors and to timestamp the monitoring data.

3 The Monitoring System Software

The monitoring system is based on the hardware described in the previous section. During monitoring, the control processors collect data about the CPU and communication link utilization for each interval, and forward this data through the control system ring to a collector process where the data is stored in a file for later processing.

The monitoring system software consists of four different parts (see Figure 6).

The *Multiplexer* process executes concurrently with the user program on each monitored target processor. It receives messages from the monitored user program each time the program communicates over a monitored link. This message contains information about which physical link the communication used, the number of bytes transfered and the total communication time. This message from the user program to the multiplexer is sent as a result of calling a pair of predefined procedures. The user has to insert these calls around each monitored communication statement in the program (see Figure 7).

The Multiplexer process receives messages about link usage from the user program and adds the number of transfered bytes and the elapsed communication time to internal variables. When it receives a sampling signal from the control system, it sends the ackumulated values for this time

```
INT StartTime :
VAL size IS ?? :
SEQ
  StartCommunication (StartTime)
  C ! X
  EndCommunication (StartTime, size, ToMux)
```

Figure 7: A monitored communication statement

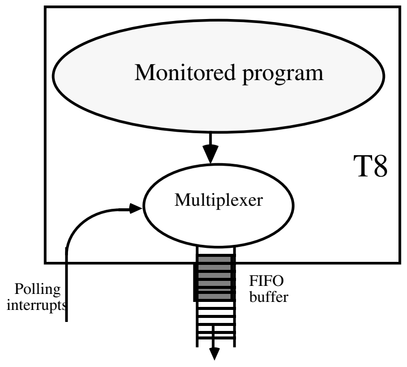

Figure 8: The Multiplexer

interval to the control system via a FIFO buffer (see Figure 8). The values that are measured for each time interval are the number of communications that has occured, the total number of bytes transfered and the total communication time for each monitored link. After that the internal variables are cleared. If there has not been any traffic on a monitored communication link during the interval, no data packet is sent.

The *Control Process* executes in each T2 processor in the control system. It reads monitoring data packets from the FIFO buffers that connect it to it's four target processors, and forwards the data packets to the Data Collector process (described below). It also reads the value of the load meter each time it receives a sampling signal, and then clears the load meter register. One of the Control Processes is also responsible for generating the sampling signals to the Multiplexer processes (see Figure 9).

The *Data Collector* receives monitoring data packets from the control system and stores the data in a file. It actually consists of two parallel processes: the first receives monitoring data packets into a large circular buffer, and the second writes the data to an ASCII file from the buffer. The circular buffer is relatively large (currently about 1.6 Mb). As long as there is room in the buffer, incoming messages can be received and stored without having to wait for disk operation, which is very slow compared to communication over transputer links.

The *Presentation Process*, which normally is executed on the user's host workstation, reads the ASCII files containing the collected monitoring data and analyses and presents this to the user in an easy-to-understand form (described in Section 4). The presentation is based on the processor structure of the monitored program and forms a replay of the program execution. The user can thus directly associate the observed utilization figures with the physical resources (CPUs and communication links) used by his program.

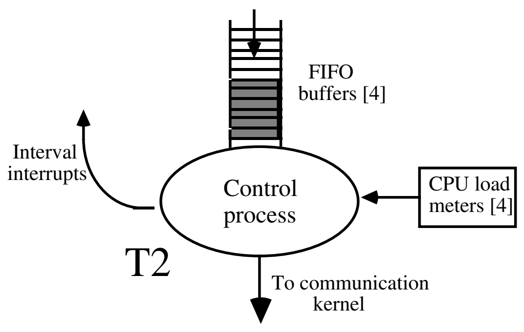

Figure 9: The Control Process

3.1 Overhead caused by the monitoring system

One important feature of all monitoring systems is that it should not change the behaviour of the monitored program. Since the system presented here is mainly intended for performance monitoring, we here only require that the monitoring of a program does not significantly change the performance behaviour of the program execution.

The overhead of the monitoring is caused by the fact that we have to insert some statements into the target program in order to obtain reports about the usage of communication links. This overhead consists of two components:

- for each monitored communication statement in the target program, a probe consisting of a pair of predefined library procedures have to be executed (see Figure 7). As a result of these procedure calls, a message containing the link number, the number of bytes transfered and the time for the communication, is sent to the multiplexer process. The time for executing these activities is denoted by T_r and has been measured to be 0.00005 s.
- for each time interval, the multiplexer process sends the values ackumulated during the time interval as a data packet through the FIFO buffer. The number of packets that is sent for one interval depends on how many communications links have been active during this interval. If no communication has taken place on a link, no data packet will be sent. In the worst case, all four input links and all four output links have been active, and 8 data packets have to be sent through the FIFO. The time for sending one packet through the FIFO is denoted by T_p and has been measured to be 0.00011 s.

From this we can see that the overhead caused by the monitoring system mainly depends on the number of monitored communications in the target program, as each communication has to be reported to the monitoring system. The interval length is of less importance for the total overhead. We can express the overhead time T_{ovh} for a monitored program execution as

$$T_{ovh} = N_c * T_r + N * T_p$$

where N_c is the number of monitored communications that take place in the monitored program, and N is the number of intervals during the program execution.

If we know the average number of communications (per second) in a program, we can for a fixed interval length express the overhead as percents of the execution time. Table 1 shows the overhead as percents of the total execution time for an interval length of 200 milliseconds.

Comm./s	Overhead (%)
10	0.06
25	0.08
50	0.10
75	0.13
100	0.16
200	0.26
500	0.56
1000	1.06
2000	2.06
5000	3.06
10000	10.06

Table 1: Overhead of monitoring for interval length 200 ms

4 Data presentation

The amount of data produced by the monitoring system can be very large. For the user to be able to interpret the data, it must be possible to reduce this large amount of information into a few relevant measures that describe the performance characteristics of the program on an appropriate level of abstraction. The results of the monitoring must also be presented in a way that reflects the structure of the monitored application, so that it is possible to associate the measured performance figures with the code of the application program. This means that when a performance bottleneck has been identified in a program, it should be easy to locate the code that caused this behaviour.

In our system, the presentation is integrated into a graphical programming environment [2], in which parallel programs are designed in the form of a *process graph*. Nodes in the graph represent processes and edges represent communication channels between processes. Processes communicate with each other by sending and receiving messages via unidirectional, synchronized, point-to-point communication channels. Each node contains the Occam code of the process.

The user interface of the programming environment consists of a graph editor, that provides tools for creating processes, connecting them together with communication channels and editing the code of the process. Processes can also be grouped together to form hierarchical process structures. A process can thus contain internal processes, which are executed in parallel. This allows the programmer to develop programs in a bottom-up fashion, and only concentrate on a small part of the program at a time.

The highest level in the process hierarchy describes the hardware structure that is needed to execute the program. In the ideal case, the process graph can be directly mapped onto a processor structure and executed. This process-to-processor mapping is done by a heuristic algorithm [13], that guarantees that a mapping always is found, possibly by merging some of the processes into one process.

The processor interconnection structure on which the program is executed is called the processor graph. The nodes in the processor graph represent physical processors (transputers) and the edges represent physical communication links (transputer links). Monitoring data from the execution of a program is presented upon this processor graph. The observed performance figures of the resources are written on the objects in the graph. In this way the user can directly associate the measured performance with his program.

The presentation system provides a set of pre-defined metrics which can be viewed in a number of different ways. CPU utilization is presented as percent of utilization during a time interval, as measured by the CPU load meter. The utilization of a link can be presented as the number of communications during an interval, or as the number of bytes transferred over the link. Furthermore, the data transmission time, the waiting time and the sum of these (the total communication time) can be presented, either as absolute values in milliseconds or as percent of the time interval.

The user can step through the performance trace of the execution one interval at a time. The system also gives the user the possibility to view mean values and standard deviations over the whole or a part of the execution for all the metrics mentioned above. Normally, the user starts by examining the average utilizations and their standard deviations over the whole program execution. If the standard deviations are very small, it is generally not necessary to step through the execution in more detail, as there are no significant variations in the resource utilizations. However, if the user wants to look at the dynamic behaviour of the program he has to step through the performance trace one time interval at a time. Figure 10 shows an example of how mean values of CPU and link utilization are presented. The program that has been monitored is the example program of Section 5, executed on 4 processors. The number in the middle of each node is the processor utilization. Link utilization is presented by two numbers X/Y, where X is the waiting time (synchronization) and Y is actual data transmission time, expressed as percents of the time interval.

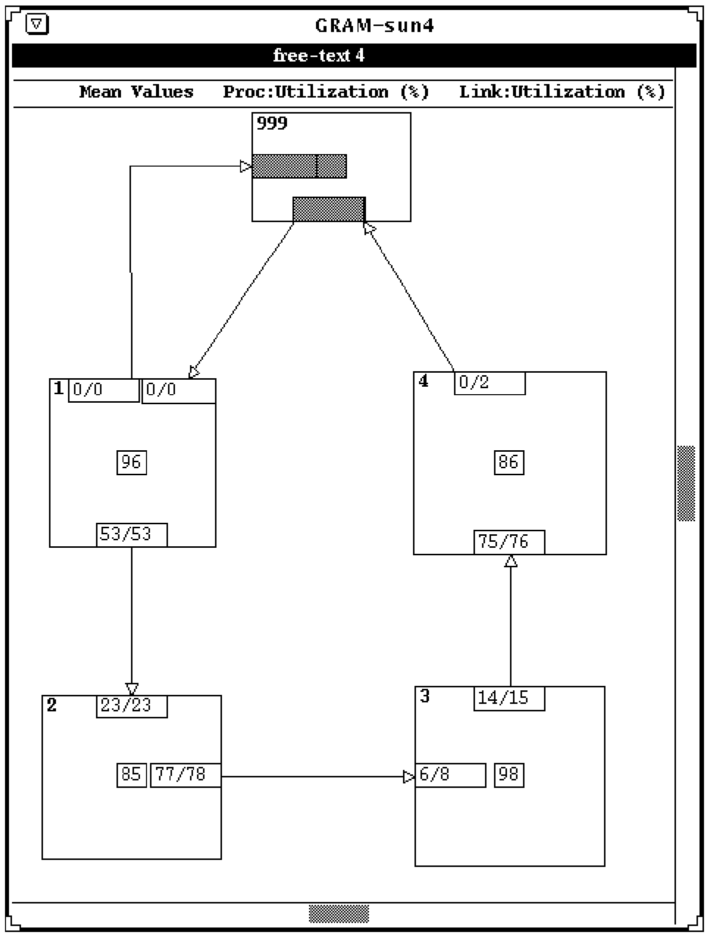

Figure 10: Monitor data presentation, showing mean values

5 A case study

In this section, we present an example of how the monitoring system can be used for analyzing and improving the performance of a parallel program. The program that we use to illustrate the ideas is a distributed free-text retrieval system [16] implemented in Occam.

The free-text retrieval system is implemented using the processor farm approach [5]. It has been mapped onto two different transputer network configurations: *binary tree* and *unidirectional ring*. For both of these configuration types, different numbers of transputers can be used. In Figure 11 an unidirectional ring[1] (which is the configuration we chose for this case study) consisting of four processors is shown. The text database is compressed and then distributed evenly among the slaves. The host in the transputer net acts as master and distributes search words to the slave processors via the first processor. The slaves do the actual text search and then forward the results to the master via the neighbours. The master recieves the answers from the last processor.

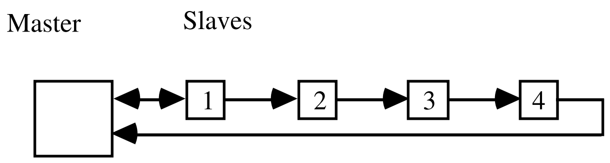

Figure 11: The structure of the free-text retrieval system when mapped onto an unidirectional ring with 4 processors

The main objective for choosing this particular program for a case study was that it has been very thoroughly tested. Because of this, the results of the case study can be related to those achieved in earlier tests, which gives us the possibility to verify the that the monitoring system functions correctly. The reason for choosing the ring configuration was that it is easy to analyze.

Table 2 shows the speedup results achieved in [16] using the ring topology. The number of transputers in the network, as well as the size of the text database, are varied. As can be seen, the speedup of the program is satisfying when 3 and 7 transputers are used. When 15 transputers are used, the speedup reaches satisfactory values when the size of the text database grows from 0.5 Mbytes towards 10 Mbytes. When the number of transputers is increased further, the speedup (and as a consequence the efficiency) drops significally, although it rises when the text database grows. Consequently, the conclusion in [16] was that the main reason for the drop of efficiency was the limited size of the text database. With a sufficciently large text database, a near linear speedup was expected. The monitoring of the program, as will be seen, suggested some additional reasons for the efficiency drop.

In the test runs we have used 4, 8, 16 and 32 transputers[2], respectively. The monitoring was done using a fixed sized text database of 0.5 Mbytes. The length of the time interval used for the measurement was 0.5 s. We used 3000 search words for each run causing execution times of a length between 30 and 90 s approximately.

All the results that are presented here are mean values over the whole program executions. The reason for this is that the utilization of different resources in this particular program vary very little. For instance, when using 16 processors the standard deviation of the utilzation of the processors and links was only a few percents. All utilization values that are presented in the tables represent the fraction of time a certain resource has been in active use (expressed as percentages of the total execution time). By link utilization we mean the combination of synchronization

[1]The observant reader might notice that there is an extra communication link from processor nr. 1 to the master. This link is only used for control purposes, mainly to terminate the program.

[2]In [16] 3, 7, 15, 31 and 63 transputers were used to make a fair comparison between the tree and ring configurations possible.

	Database size (Mbytes)			
Nr. of processors	**0.5**	**1**	**3**	**10**
3	3.00	-	-	-
7	6.32	6.80	-	-
15	11.15	12.73	14.00	14.23
31	9.74	11.38	13.01	13.69
63	5.27	5.38	6.00	6.14
96	4.35	4.44	4.73	5.03

Table 2: Speedup for original program using ring configuration

and actual data transmission. The term waiting means the fraction of time spent waiting for synchronization.

When optimizing parallel programs on a transputer network one would like to see the CPUs being utilized as much as possible. If their utilization is low, it is helpful to know the reason for this. At this stage it is also interesting to observe the behaviour of the communication links. Of interest is, for instance, if some of the communication links on the transputer whose CPU is poorly utilized, are in heavy use. This could indicate that the link in case is a physical bottleneck limiting the performance. On the other hand, if the amount of time spent waiting for synchronization on a particular link (especially if it is an output-link) is very large, then the processor on the other end of the link could be considered a reason for possible inefficiency. Generally, this type of inefficiency is caused by two major factors: either the processor on the other end of the connection is too slow (i.e. it is a physical bottleneck) or then the process running on it does not use the hardware capabilities of the transputer (e.g. DMA) very well.

	Link utilization		
Nr. of processors	**Input links**	**Output links**	**CPU utilization**
4	28.1	27.5	90.8
8	11.6	53.0	87.9
16	13.0	71.8	60.5
32	13.1	81.8	36.1

Table 3: Mean values of the utilization of all CPUs and communication links

In Table 3 mean values for utilization of all CPUs and all input and output links are shown. We can see that when the number of transputers in the ring is increased, there is a considerable imbalance in the utilization of resources. For instance, when 32 processors are used, the mean utilization of the CPUs is only 36.1 % and for the input-links it is even lower (13.1 %), but the output-links are in heavy use.

From the table above, it is hard to get a more exact idea of what the problem is. The picture becomes clearer when we analyze Table 4, where we look at the first and the last processor in the ring. This data show that the utilization of the output-links is much smaller for the last processor than for the first. Also the time spent waiting for synchronization is smaller. In fact, there is a clear trend of decreasing utilization and waiting as we move forward in the ring. This can be seen clearly when the monitoring data is presented in the presentation environment, where one can observe the whole network at a time. The data suggest that there is a queue of processors that want to send data forward in the ring, but are unable to do this because the next processor

	First transputer in ring		Last transputer in ring	
Nr. of processors	**Utilization**	**Waiting**	**Utilization**	**Waiting**
4	49.7	48.9	6.8	0.6
8	72.3	70.8	21.8	2.6
16	85.8	84.1	54.4	15.5
32	91.7	90.6	70.2	23.2

Table 4: Mean value of utilization and waiting time (%) of output-link on first and last transputers

in the ring is in the same situation and therefore cant recieve the data.

It is quite clear that if the waiting time on the last output-link is eliminated then the throughput will be increased. When examining the code, we discovered that there was a small logical error in the master process, which was the reason for the long synchronization times. The major reason why this error never was discovered was that the author did not know of the fact that there was an extra waiting time involved. Therefore the conclusion was that the efficiency drop was caused by a text database that was too small.

In the test runs described above a T414-transputer was used as the master, while all processors in the ring are of the T800-series. Since the data communication protocol used by the T414 is less efficient than the one used by the T800, the throughput could also be increased by replacing the T414 by a T800, and thereby limiting the data transmission time between the last processor in the ring and the master processor.

We eliminated the logical bug and moved the program to another host computer where a T800 was used as a host processor of the transputer net. These changes improved the performance of the program in a way presented in Table 5. As can be seen, the result does not change for the already efficient ring configurations using 4 and 8 processors. For the larger configurations, especially the one using 32 processors, the improvements are apparent.

	Speedup	
Nr. of processors	**Original program**	**Enhanced program**
4	4.00	4.00
8	7.58	7.71
16	10.10	13.25
32	5.27	17.23

Table 5: Speedup for original and enhanced versions of program

The improved program has also been monitored. The results in Table 6 indicate that for the configurations using up to 16 processors the load is quite well balanced. However, Table 6 gives a hint how the performance of the configuration using 32 processors could be improved even further. As can be seen, there is still an imbalance between the utilization of the processor in the beginning and the end of the ring. This is due to the fact that each processor in the ring has an equally large part of the text database to inspect. Because communication in the ring is unidirectional, the amount of time spent forwarding messages (results) will be significant for the processors in the end of the ring. Therefore they will devote a big part of their time to handling these messages. As a consequence, the throughput of the system will decline and the CPU utilization on the processors in the beginning of the ring will fall. A better load balancing could actually solve this problem. This could be achieved for instance by using a bidirectional ring configuration, or by distributing differently sized parts of the text database so that the total amount of work (including message forwarding), instead of the size of the text database, would be even.

	CPU utilization	
Nr. of processors	**First processor**	**Last processor**
4	97.8	86.3
8	94.4	86.4
16	82.5	86.3
32	57.4	89.3

Table 6: Mean value of CPU utilization for first and last processor in enhanced program

6 Conclusions and future work

The monitoring system for Hathi-2 has shown to be a useful tool. According to our experience, it can successfully be used to locate performance bottlenecks in parallel programs. For instance, the free-text retrieval system used for a case study in this paper, was considerably improved using the tools described here. Also, it gives the programmer good insight in how a parallel program behaves and how well it uses the hardware resources of the multiprocessor system during execution.

Including the monitoring system in a graphical programming environment seems to be a good soultion, as it gives the user the ability to view the results upon the processor structure he has designed himself. This is a considerable improvement over the traditional aproach using tables and histograms.

Currently, the software probes have to be inserted by hand. This task can be very laborious and will be automated in the future. A problem is that for some programs, the intrusiveness of the monitoring system becomes unacceptable. This is the case for programs that send or receive a very large number of messages. To avoid this, there are plans to implement a sampling monitor for the monitoring of communication links. For the future, there are also plans to implement a high-level debugging and animation system, based on the same concepts as the current monitoring system.

Acknowledgements

This work has been done in the FINSOFT III research program, which is financed by TEKES. The monitoring data presentation software was written by Henrik Gullberg. The initial version of the monitoring system software was written by Stefan Levander and Tor-Erik Malén.

References

[1] T. Äijänen. Distributed interconnection of a reconfigurable multicomputer system. *Microprocessing and Microprogramming*, (3):243–246, 1988.

[2] M. Aspnäs and R.J.R. Back. A programming environment for a transputer-based multiprocessor system. Reports on Computer Science and Mathematics Ser. A 82, Åbo Akademi, 1989.

[3] M. Aspnäs, R.J.R. Back, and T-E. Malén. Hathi-2 multiprocessor system. *Microprocessors and Microsystems*, 14(7):457–466, September 1990.

[4] D. Haban and D. Wybranietz. A hybrid monitor for behaviour and performance analysis of distributed systems. *IEEE Transactions on Software Engineering*, 16(2):197–211, February 1990.

[5] A.J.G. Hey and D.J. Pritchard. Parallelism in scientific programming and its efficient implementation on transputer arrays. Technical report, Concurrent Computation Group, Department of Electronics and Computer Science, University of Southampton, 1987.

[6] Jeffrey Joyce, Greg Lomow, Konrad Slind, and Brian Unger. Monitoring distributed systems. *ACM Transactions on Computer Systems*, 5(2):121–150, May 1987.

[7] Teemu Kerola and Herb Schwetman. Monit: A performance monitoring tool for parallel and pseudo-parallel programs. *ACM Sigmetrics Performance Evaluation Review*, 15(1):163–174, May 1987.

[8] A. E. Knowles and M. S. Illiev. Monitoring facilities on the ParSiFal T-Rack. In *Proc. CONPAR-88*, pages 49–55, 1988.

[9] Ted Lehr, Zary Segall, Dalibor F. Vrsalovic, Eddie Caplan, Alan L. Chung, and Charles E. Fineman. Visualizing performance debugging. *IEEE Computer*, pages 38–51, October 1989.

[10] Inmos Limited. *Transputer Reference Manual*. Prentice–Hall, 1988.

[11] Barton P. Miller. DPM: A measurement system for distributed programs. *IEEE Transactions on Computers*, 37(2):38–51, February 1989.

[12] R. Pooley (ed.). Introduction to the second Posie report. Internal report CSR-6-90, University of Edinburgh, Department of Computer Science, 1990.

[13] H. Shen. Self-adjusting mapping: a heuristic mapping algorithm for mapping parallel programs onto transputer networks. In J. Wexler, editor, *Developing Transputer Applications (Proc. OUG-11)*, pages 89–98. IOS, 1989.

[14] Richard Snodgrass. A relational approach to monitoring complex sytems. *ACM Transactions on Computer Systems*, 6(2):157–196, May 1988.

[15] Jeffrey J.P. Tsai, Kwang-Ya Fang, and Horng-Yuan Chen. A noninvasive architecture to monitor real-time distributed systems. *IEEE Computer*, pages 11–23, March 1990.

[16] M. Wallde'n and K. Sere. Free text retrieval on transputer networks. *Microprocessors and Microsystems*, pages 179–187, April 1989.

[17] Cui-Qing Yang and Barton P. Miller. Performance measurement for parallel and distributed programs: A structured and automatic approach. *IEEE Transactionson Software Engineering*, 15(12):1615–1629, December 1989.