Multiprocessor Applications in the Hathi Project

M. Aspnäs * R.J.R. Back *

November 7, 1989

Abstract

This report describes some of the more important applications of parallel processing in the Hathi project. The aim of the applications were to evaluate the use of transputer-based parallel processors for solving large computational problems. The report presents the problems solved in the application programs and the experiences gained from this work.

1 Introduction

This report describes a collection of application programs designed for the Hathi-2 multiprocessor system in the Hathi project. Most of the applications in the project were carried out in cooperation with other research institutions, but some were internal to Åbo Akademi. The problems solved in the applications have been choosen to be demanding computational problems, in which a very large processing power is required and where good results from parallel execution could be expected.

One goal of the application programs was to show that parallel processing can be successfully used for solving very large computational problems. Another goal was to collect experience from parallel programming in order to identify the problems with this technology.

This paper is organized as follows. The Occam language, in which most of the applications on Hathi-2 were programmed, is presented in Section 2. The architecture of the Hathi-2 multiprocessor is presented in Section 3. Section 4 describes the main applications implemented on Hathi-2. In Section 5 the experiences from this work is summarized.

2 The Occam programming language

Most of the applications for Hathi-2 were programmed in Occam, a language specially designed for parallel programming. Some of the applications were written in Parallel Fortran, mainly because existing sequential code could be reused in the parallel program and this language was more familiar to the people involved in these applications. Both these programming languages contain similar primitives for expressing parallelism and communication, except that in the case of Occam the language itself contains these primitives and in other scientific languages (like Fortran and C) these primitives are added to the language in the form of system functions that can be called from a program. Occam is breifly presented here as an exaple of a language for parallel programming.

Occam [Inmos 88b, Jones and Goldsmith 88] is a high-level programming language based on the CSP language [Hoare 78]. An Occam program consists of a number of sequential processes, which are executed in parallel and communicate with each other via unidirectional channels using

*Åbo Akademi, Department of Computer Science, Lemminkäisenkatu 14, SF-20520 Turku, Finland

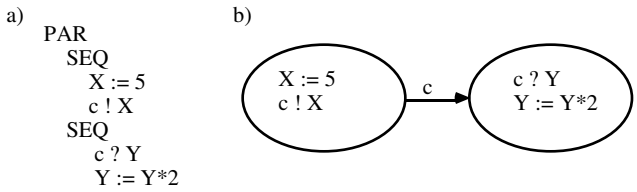

Figure 1: Communicating processes in Occam, a) Occam code b) pictorial representation

synchronous message passing. Occam contains most of the constructs found in Pascal, like data types, assignment-, IF-, FOR- and WHILE-statements etc. However, variables of pointer type, dynamic creation of variables and recursion is not supported in Occam.

An Occam channel connects two processes, of which one acts as a sender and the other as a receiver. A process sends a message *M* via a channel *c* with an output statement *c!M*, and the receiving process inputs a message from the channel to a local variable with an input statement *c?M*. A process can wait to receive input from a number of channels at the same time, using an **ALT** construct. The sending process can not choose between different communication alternatives, but commits itself to communication on a specific channel when it executes an output statement. Communication is synchronous, i.e., the process which first executes a communication statement remains waiting until its communication partner executes a corresponding communication statement.

Parallelism is expressed in Occam by the **PAR** construct, which specifies that two or more processes are executed in parallel. Sequential execution is specified with the **SEQ** construct. Scope is expressed in Occam by indentation. In the example in Figure 1, the two processes communicate with each other via a channel *c*.

More than one process can be executed simultaneously on one transputer. The transputer divides its time between processes using a simple round-robin scheduler which is built into the transputer hardware and thus very efficient. Communication between processes executed on the same transputer is implemented through direct memory access, while communication between processes executed on different processors takes place via transputer links.

To execute a program with real parallelism on more than one transputer, the programmer has to describe on which transputers the processes are to be executed and which communication links are used for communication between the processes. The user has to explicitly describe on which processor each process is executed and which communication links are used for communication between the processes. This is done by an Occam-like configuration language. The example in Figure 2 describes a ring of three processors, each executing a process called Calculate. The processes communicate with each other by inputting from link 3 and sending on link 2.

3 The Hathi-2 Multiprocessor System

Hathi-2 is a reconfigurable general purpose multiprocessor system consisting of 100 32-bit IMS T800 floating point transputers, 25 16-bit IMS T212 transputers and 25 IMS C004 crossbar switches [Inmos 88a]. The system can be characterized as a loosely coupled MIMD multiprocessor, with a reconfigurable distributed interconnection network and a modular design. A more detailed description of the Hathi-2 architecture can be found in [Aspnäs et al. 89] and [Pehkonen 89]. The distributed switching network is described in [Äijänen 88]. The use of the Hathi-2 system is described in [Aspnäs and Malén 89].

The parallel computational power of the present system is 150 MFLOPS. Each of the 100

a)

```
CHAN OF INT C0, C1, C2:

... SC PROC Calculate (CHAN OF INT From.previous, To.next)

PLACED PAR

PROCESSOR 0 T8
PLACE C0 AT link2out :
PLACE C2 AT link3in :
Calculate (C2, C0)

PROCESSOR 1 T8
PLACE C0 AT link3in :
PLACE C1 AT link2out :
Calculate (C0, C1)

PROCESSOR 2 T8
PLACE C1 AT link3in :
PLACE C2 AT link2out :
Calculate (C1, C2)
```

b)

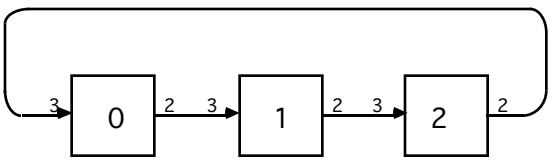

Figure 2: Placing processes on processors, a) Occam configuration code b) pictorial representation

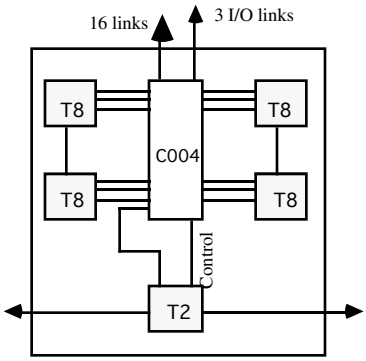

Figure 3: Hathi-2 board architecture

computing transputers has 1.25 Mb of local memorygiving a total central memory of 125 Mb.

The architecture of the Hathi-2 multiprocessor system was specified jointly by the Technical Research Center of Finland (VTT/TKO) in Oulu and Åbo Akademi, while the hardware design and construction was done by the former. The system software for Hathi-2 has been constructed at Åbo Akademi.

Hathi-2 board Hathi-2 consists of 25 identical boards, each containing four T800 transputers, one T212 transputer and one 32 link crossbar switch. The T800 transputers are connected pairwise to each other via one of the four communication links. The three remaining links are connected to the crossbar switch (see Figure 3). Three links from each switch are used as I/O links, i.e., to connect users host computers and peripheral units to the system. The remaining 16 links from the crossbar switch are used to connect the transputers on the board to transputers on other boards.

The C004 crossbar switch is controlled by the T212 transputer via a control link. Another link on the T212 is connected to the crossbar switch and can be connected via the switch to any other transputer link. The two remaining links on the T212 are used to connect the T212 transputers into a ring, thus forming the distributed control system.

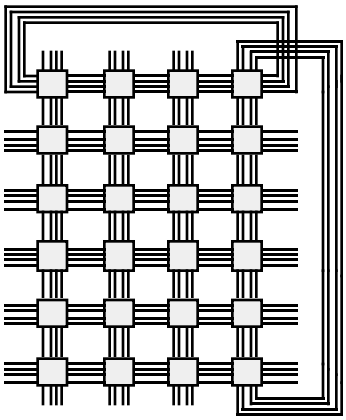

Figure 4: Hathi-2 board connections (some wrap-around links omitted)

The switching network The crossbar switches on the Hathi-2 boards are connected to each other in a static torus connection. Each pair of neighbouring boards are connected by four links (see Figure 4). The crossbar switches form a distributed switching network connecting the communication links of the T800 transputers, which enables the system to be reconfigured by software.

Use of Hathi-2 Hathi-2 is used as a back-end computing resource. The user edits, compiles and links his programs on a host computer, i.e., a Sun workstation. The program can then be loaded on to the multiprocessor system and executed.

The Hathi-2 system can be shared between a number of simultaneous users by partitioning it into several smaller independent multiprocessor systems. A user is allocated a separate partition which is independent of all other partitions. The user has full control over his own partition, but can not interfere with other users.

Hathi-2 is connected to a Sun 3/160 workstation, which in turn is connected to the local Ethernet network in Åbo Akademi (see Figure 5). This network can be accessed from the national network connecting the Finnish universities, as well as from other international networks.

The control system The T212 transputers are connected to each other in a ring, thus forming a separate control system which controls the switching network (see Figure 6). The control system is totally independent of the rest of the system. The only connection between the user and the control system is via a link connecting one T212 transputer to the users host computer. The user can request system services by sending commands to the control system via this link.

The control system has two main tasks: to control the distributed switching network and to monitor the activities in the system. The Hathi-2 architecture contains hardware dedicated to monitoring the resource utilization in the system. The monitoring hardware consist of a CPU load meter which measures the CPU utilization by observing the bus activity and a FIFO buffer connecting all T800 transputers on a board to the controlling T212 transputer. The FIFO buffer can be used for sending reports about resource utilization from the T800 to the T212 without affecting the communication links.

The control system also contains an interrupt subsystem implemented using the transputers EVENT interrupt. A processor in the control system can send an interrupt signal to all processors in the same partition. This interrupt is used in the monitoring system to generate a synchronizing

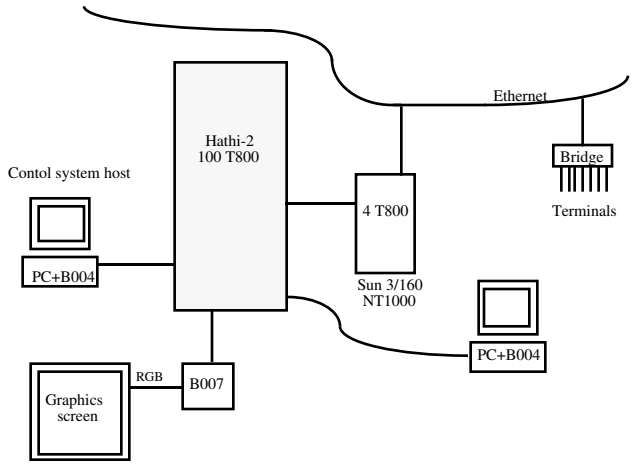

Figure 5: The environment in which Hathi-2 is used

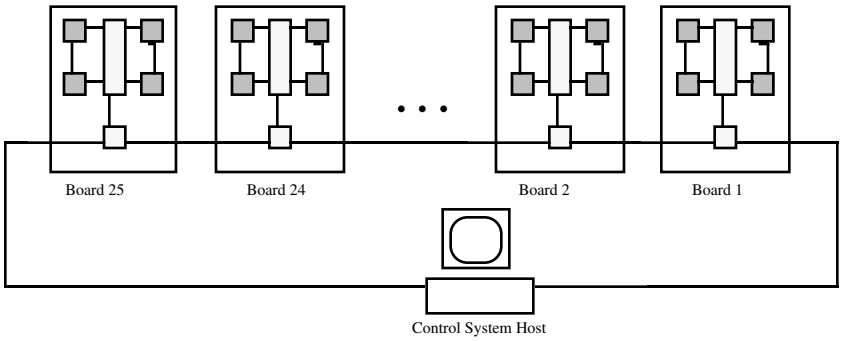

Figure 6: The control system

signal which divides the time into short time intervals. The CPU and link utilization are measured for each interval and reported to the user.

Scalability The hardware design for Hathi-2 is easily scalable. A much bigger Hathi-2 system can be built by simply using more of the Hathi-2 boards. A 1000 processor system could thus be built with 250 of these boards and would give a total parallel efficiency of 1.5 GFLOPS with todays hardware. The system is also relatively cheap. Because of the distributed architecture, all the boards are identical, and the cost of the board is dominated by the components, i.e. the processors (T800 and T212) and the memory. This means that the hardware cost of a Hathi-2 multiprocessor system is more or less linear in the number of processors, the amount of memory and the number of peripheral units that are connected to the system.

System software The system software developed for Hathi-2 is concentrated on two main areas: configuration software to control the distributed switching network and monitoring and animation software to observe the behaviour of a parallel program executing on the system.

The configuration software enables the user to reconfigure the processor interconnection structure of his partition to almost any structure. Among the most frequently used processor inter-

connection structures are torus, grid and tree structures. The monitoring software gathers information about the resource utilization (i.e. the utilization of the CPUs and the communication links) of the system during the execution of a parallel program. This utility is used to identify bottlenecks in the computation and thus improve the load balance of the program.

The system software for Hathi-2 executes on the control system, which is dedicated to this purpose. The system software utilities are all based on a general message-passing system implemented on the control system. The message-passing system consists of a small communication kernel process executing on all control processors. The communication kernel handles transparent message passing between the processes in the control system. To this message passing kernel, a number of *service processes* can be attached. These service processes provide the above described services, i.e., reconfiguration and monitoring, to the user.

Most commercially available software for transputer based systems can also be used on Hathi-2, like for instance Occam, C and Fortran compilers and the distributed operating systems Helios [Perihelion 89] and Trollius [Burns et al. 88].

4 Multiprocessor applications

One of the main goals in the Hathi-project was to experimentally try out the efficiency of parallel computation in practical applications. To this end, a number of application projects were initiated, some internal to the department and some in co-operation with other research institutes. Most of these applications turned out to be quite successful and deliver the efficiencies that were expected. The applications were initially carried out on a smaller 16 transputer system (Hathi-1), and then ported to the larger Hathi-2 when it became ready, in order to measure speedups and efficiencies when more masssive parallellism was available. Some of the applications are now being developed further as independent projects, for instance the fluid dynamic modelling and the real-time transformation of satellite pictures. Four of these applications are described in the sequel in more detail.

Programming multiprocessor applications An efficient implementation of a computational problem on a MIMD-type multiprocessor system does not necessarily follow the same ideas as an implementation on a sequential processor. Generally, it is not possible to construct a parallel solution using the same methods as in a sequential solution. To write a parallel implementation of a problem on a multiprocessor system, the programmer needs insight in the problem to be able to decompose it into a number of parallel processes.

All communication in the parallel solution of a problem can be considered as overhead introduced by the decomposition. On most existing multiprocessor systems, communication is slow compared to computation. To get an efficient parallel implementation, the processor must therefore perform a sufficient amount of calculation for each communication. Generally, this means that one should try to avoid communication in parallel algorithms, even at the cost of additional computation.

Let the execution time for a program executed on N processors be denoted by T_N. The speed-up factor S_N for a parallel program executed on N processors is then defined as

$$S_N = \frac{T_1}{T_N}$$

and the efficiency E_N of a parallel program executed on N processors is

$$E_N = \frac{S_N}{N}$$

The sequential program that we compare the parallel program to should be implemented using the best known sequential algorithm for the problem. This is not always possible as it might require an extensive amount of programming. At the very least, the sequential program should not be constructed by executing the parallel algorithm on one processor using timeslicing between the parallel processes. In that case, the execution time for the sequential algorithm will contain overhead caused by the scheduling of the parallel program on one processor, and the measured speed-up will not be accurate.

In the ideal case, the speed-up S_N is equal to N, the number of processors used. More important is that the speed-up grows linearilly with an increasing number of processors. However, when we increase the number of processors, the amount of work per processor will decrease, wich will have a negative effect on the speed-up. When increasing the number of processors we must therefore also increase the problem size, so that the amount of computation per processor remains constant. This means that the problem should be big enough in order to achieve an efficient parallel implementation. This fact can be clearly seen in the applications described below. For a fixed problem size, there exists a limit where additional processors does not result in any better performance. However, if the size of the problem can be scaled up, by increasing the amount of data or by increasing the quality of the solution, linear speed-ups can be acheived for a large class of computational problems.

4.1 Chemical reactor, heat transfer and fluid flow modelling

The chemical reactor, heat transfer and fluid flow modelling applications were developed in co-operation with the Process Design and the Heat Engineering Laboratories in the Department of Chemical Engineering at Åbo Akademi. The simulation of a chemical reactor was carried out by Tom Björkholm (M.Sc. Thesis, Eng.), who used the so called processor farm approach. See also [Kilpinen et al. 89]. A set of heat transfer and fluid flow problems was solved by Pekka Kuusela (M.Sc. Thesis), Tor-Erik Malén and Göran Öhman [Öhman et al. 88].

Problem description The reactor process modelled was a packed-bed two phase (gas-solid) chemical reactor for iron ore reduction. One of the objectives was to design transient (batch) experiments in order to obtain accurate parameter estimates in the reduction kinetics model. Another objective was to use the model for on-line simulation of blast furnace.

Simulation of the packed bed reactor involves solution of 30 ordinary differential equations (ODEs). The CPU time consumed on a micro VAX II computer is on an average between one and ten minutes depending on the problem formulation.

The heat transfer and fluid flow study started from the numerical solution of the two-dimensional Laplace equation, which describes the steady heat conduction in a solid plate, and advanced through the solution of the three-dimensional Laplace equation to the case of stedy laminar fluid flow in a two-dimensional box at Reynolds numbers up to 20. Hereby the stream function-vorticity method was first applied and then the SIMPLER method.

The essential principles which were used in the parallel solution of these problems are illustrated by the physically simplest case, the two-dimensional heat transfer problem. Consider a solid square plate of size x_0 by x_0 which is thermally insulated on both sides. The temperature along the edges of the plate is given and is assumed to be fixed. The problem is to calculate the steady temperature distribution in the interior part of the plate. The problem is illustrated in Figure 7.

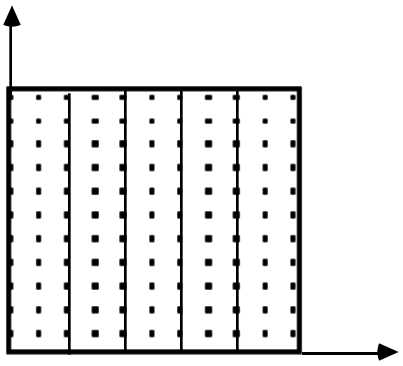

Figure 7: The discretized temperature field partitioned into slices

Mathematical model Mathematically, the steady temperature distribution $\theta(x, y)$ in a homogeneous solid plate is described by the Laplace differential equation

$$\frac{\partial^2\theta}{\partial x^2} + \frac{\partial^2\theta}{\partial y^2} = 0$$

for $0 \leq x \leq x_0, 0 \leq y \leq x_0$. The solution must also satisfy the initial boundary conditions describing the temperature along the edges of the plate.

This Laplace differential equation can be solved numerically by discretizising the field and calculating the temperature only in a finite set of points on the plate. Choosing an equal spacing h in both x and y directions, the temperature in an arbitrary point (x, y) is expressed as an algebraic equation

$$\theta(x, y) = [\theta(x + h, y) + \theta(x - h, y) + \theta(x, y + h) + \theta(x, y - h)]/4$$

The equation is transformed into a dimensionless form by introducing the dimensionless coordinates $X = x/h$ and $Y = y/h$. It can then be rewritten as

$$T(X, Y) = [T(X + 1, Y) + T(X - 1, Y) + T(X, Y + 1) + T(X, Y - 1)]/4$$

i.e., at each internal point in the plate the temperature must be equal to the arithmetic mean of the temperatures at its four neighbouring points.

The equation is solved numerically using the Gauss-Seidel method with overrelaxation, where the temperature of a point (X, Y) in iteration step $i + 1$ is calculated by

$$T_{i+1}(X, Y) = \omega(\sum T^{nb})/4 + (1 - \omega)\, T_i(X, Y),$$

where $\sum T^{nb}$ is the sum of the most recent temparature values in the four neighbouring points and ω is the overrelaxation factor. The temperature for each point in the plate is calculated iteratively until a sufficient number of iterations has been performed. Initially, each point is given a guessed initial value, often chosen to be 0. The accuracy of the solution depends on the number of iterations and the spacing h.

Parallel solution The problem was solved using *geometrical parallelism*, where the data domain of the problem is distributed among the processors and all processors execute identical code. The matrix describing the temperatures in the discrete points of the plate is divided into N equally large slices in the X-dimension, where N is the number of processors.

For all points that are not on the border between two slices, the processor holding this part of the matrix can compute the new temperature during an iteration step without communicating with any other processor. In order to calculate the temperature of a borderpoint, a processor has to excange information about the temperatures on the border line. Each processor has a copy of the neighbouring processors border values. For each iteration, the processors exchange the border values and then compute the new values for its own grid points.

Because a processor only needs to communicate with two neighbours, the processors were arranged in a ring. To be able to overlap communication with computation, high priority buffer processes were introduced, which take care of the communication independently of the calculating process. The root processor does not participate in the calculation, it only initiates the computation by sending out the initial values and accepts the results after the computation has finished.

Performance The algorithm was executed on 4, 8, 12 and 16 processors with a grid size of 20∗20, 40∗20, 60∗20 and 80∗20 points respectively. The number of iterations varied from 20 to 1000. The results of the test runs showed an almost linear speed-up. The efficiency varied between 99 % (for four processors) and 80 % (for 16 processors). The tests also showed that by increasing the grid size and the number of iterations, the speed-up is approaching N, the number of processors.

This shows that these types of problems can be solved very efficiently on MIMD-type multiprocessor systems, given that the problem is large enough to be partitioned among a number of processor.

4.2 Real-time transformation of satellite pictures

The real-time satellite data transformation application was developed by Atte Kortekangas, Aarne Rantala, Antti Raunio and Dan-Johan Still [Rantala et al. 88, 89]. The problem originates from a project carried out jointly by the Tecnical Research Centre of Finland (VTT/TIK), Vaisala OY and the Finnish Meteorological Institute.

Problem description A polar orbiting NOAA-series satellite, used for weather forecasting, takes pictures of Scandinavia and sends the picture data down to the earth, where it is received. The received pictures are distorted due to the curvature of the earth, the eccentricity of the satellite orbit, the varying viewing angle of the camera etc. From the received raw data, the users want to produce pictures in some known cartographic projection, e.g. polarstereographic projection. The transformation from raw satellite data to a cartographic projection is a very computationally intense task.

Data is received from the satellite at a rate of about 133 Kbytes/s during an overflight, which as a maximum lasts about 12 minutes. The total amount of data received during an overflight can be up to about 115 Mbytes. A real-time system, i.e., a system that performs the transformation at the same time as the data is received, should be able to produce transformed images at a rate of about 88 Kbytes/s. In this application, the problem size is fixed and can not easily be scaled up. However, more important than to acheive a good speed-up in a parallel solution is to fulfill the stated real-time requirements.

Sequential solution Let the original picture received from the satellite be denoted by a matrix Q of pixel values and the transformed picture to be computed by a matrix P. The transformation from Q to P can be described by a set of pixel tuples $(x_q, y_q), (x_p, y_p)$, stating that the pixel positioned in (x_q, y_q) in the original data corresponds to the pixel in (x_p, y_p) in the transformed

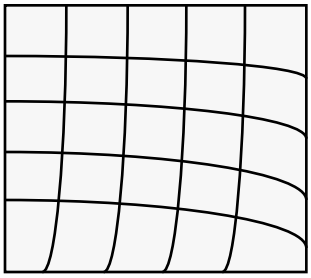

Figure 8: Raw satellite data covered by the grid

picture. The numerical value of the pixel does not change in the transformation, only the position of the pixel in the picture.

Because of neighbourhood preserving properties in the transformation, it is sufficient to calculate this exact correspondance between pixels in the raw data and pixels in the transformed image only for a small number of pixels, which form a sparse grid covering the data. The rest of the pixels are approximated by interpolation. The method is illustrated in Figure 8.

As soon as the orbit parameters for the overflight are received in form of a telex, the transformation for the pixels in the grid can be calculated. The grid size in our example is 32∗32 pixels.

Parallel solution A prototype version of the satellite data transformation system, with a much smaller amount of data and artificially generated distortions, was implemented in the project as a *processor farm*. A processor farm consists of a master processor who divides the task between a number of slave processors. The slaves all execute the same code: they receive a subproblem from the master, solve this subproblem and send the results back to the master, upon which they receive a new subproblem. The master acts only as an administrator who distributes the problem to the slaves.

The master processor holds the original distorted picture Q which it has received from the satellite. It divides the matrix into a number of submatrixes by placing a sparse grid onto the picture. For each predefined grid point (x_g, y_g) in Q, it computes the corresponding position in the transformed picture P. The transformation function is determined by the satellite orbit parameters. This computation can be started as soon as the orbit parameters for the overflight are available and does not have to take place in real-time, during the overflight.

The transformation problem is partitioned into a number of independent subtasks which can be solved in parallel. A subtask consists of the four corner points of the grid, their coordinates in the transformed picture and the pixels in the distorted picture that fall inside these grid points. The processors are connected to each other in a one-dimensional array. The master sends output packets, consisting of the data associated with one subproblem, and accepts result packets. The slaves execute three simultaneous processes, as illustrated in Figure 9:

1. a communication process that handles communication to the processor and forwarding of messages to other processors

2. a pixel mapping process, that based on the earlier calculated transformed values for the corner points in a grid block interpolates the values of the rest of the pixels in the block

3. a gathering process, that collects pixel lines from the mapping process into complete rectangular output blocks, which are sent back to the master.

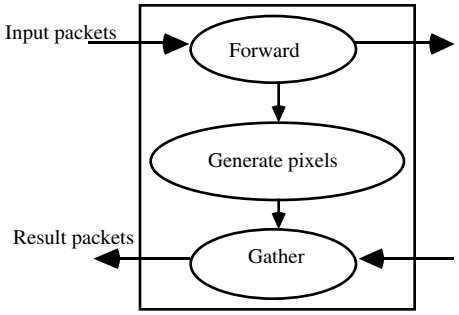

Figure 9: Process structure of the slaves

Processors	Without gather	With gather
1	7.3	8.7
2	3.9	5.0
4	2.3	3.3
8	1.5	2.4
16	1.3	2.2

Table 1: Processing times in seconds for 512∗512 pixels

The gather process is needed when large transformed pictures are to be written to a disk. If the picture is large, the whole picture can not be stored in main memory but has to be written to a disk in reasonable large blocks. If the picture is small it can be stored in the main memory of the master processor and written to the disk when the whole picture has been assembled.

Results The system was tested with artificially generated distorted pictures. The image size used in the tests is 512∗512 pixels and the number of processors varies between 1 and 16. The results of the tests are presented in Table 1. Time is measured in seconds and represent the time taken to transform one picture of 512∗512 pixels.

Tests have been carried out both with and without the gathering phase. As can be seen, the gathering causes a significant overhead on the computation. However, for full-scale satellite pictures, this phase is necessary as the picture has to be written to a disk in smaller blocks.

In the best case (using 16 T800 transputers), transformed images was produced at a rate of about 116 Kb/s, which clearly satisfies the real-time requirements of about 88 Kb/s.

4.3 Three-dimensional cluster identification in nuclear accelerator data

This application was done in co-operation with the Department of Physics at the University of Jyväskylä and the department of Physics at Åbo Akademi, by Ralph-Johan Back, Jens Granlund, Jorma Hattula, Tom Lönnroth and Patrick Waxlax.

Problem description The object of the study was to help in data analysis for nuclear physics experiment carried out on a nuclear accellerator at the University of Jyväskylä. A spectrometer detecting gamma radiation produces 10^8 to 10^9 observations at a rate of 2000 to 5000 observations per second. Each observation consists of a coordinate in a three-dimensional space of size 4096∗4096∗4096. A large amount of the observations is randomly distributed noise, while the rest

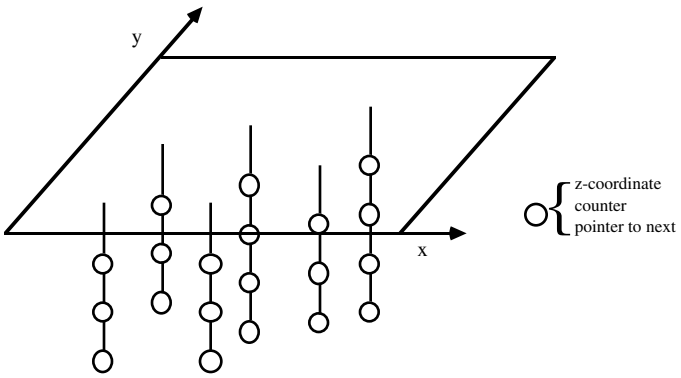

Figure 10: Data structure for storing observations

of the observations, representing actual physical events, form clusters in the observation space. A cluster is defined as a concentration of observations which contain at least n observations within a radius of r units in the three-dimensional space, for some given values of n and r. Of the incoming observations, up to 90% can consist of noise and the remaining 10% correspond to data from real observed events. The problem is to identify the clusters, their position in the space and their intensities, i.e, the number of observations in the cluster.

Solutions The problem in this application is that the amount of data is very large. If all observations could be stored in a matrix of size 4K∗4K∗4K, the solution would be straightforward. However, this is not possible as it requires 64 Gbytes of memory to store the data in.

An alternative solution is to store the observations in a bit-map of 4K∗4K∗4K bits, which would reduce the memory requirement to 8 Gbytes. The bit-map would be very sparse, as no more than 1.5 percent of the entries would contain any observation. It must also be possible to register more than one observation in one position, which would require additional memory and also would complicate the algorithm. This solution was therefore also rejected because of the large memory requirement.

In the solution adopted, the observations are stored in a two-dimensional array consisting of linked lists of observations. Each element in the array consists of the (x, y) coordinates of the observations and a pointer to a list of z-values. The elements in the z-list consist of the z-coordinate, a counter indicating the number of observations in this point and a pointer to the next observation with the same (x, y) coordinates. The data structure is illustrated in Figure 10.

With this representation, one pointer for each (x, y) coordinate pair is needed, i.e., 4∗4K∗4K = 64 Mbytes. For each observation, one has to store the z value, the counter and a pointer to the next observation, giving a total of 7 bytes per observation. For 10^8 observations, the maximum memory requirement will be about 764 Mbytes. However, measurements showed that the average number of observations for the points that occur in the observations is between 4 and 5, so the actual memory requirement can be reduce by a factor of 4, giving a total memory requirement of about 190 Mbytes.

Implementation Because of the limited amount of memory in Hathi-2 at the time when the application was designed, the implementation was scaled down to a size of 400∗400∗4000 points. The problem was decomposed using geometrical parallelism, where each processor is responsible for a part of the space. The data domain is divided in equally large blocks in the (x, y) plane. Each processor keeps record of the observations falling inside its own domain, and whenever a new

observation arrives, the processor inserts the observation into the appropriate z-list and checks, by searching the z-lists of the neighbouring points, if a cluster was formed. When a cluster is found, the observations belonging to this cluster are removed from the z-lists and only the position and intensity of the cluster is stored in a separate list. New observations are also checked against this cluster-list to see if the observation falls into some already detected cluster.

The processors are connected to each other in a ring. The host processor sends the observations in form of (x, y, z) coordinates to the slaves. When a slave receives an observation, it checks wether the observation belongs to its own domain, in which case it processes the observation, or if it should send the observation to the next processor in the ring. In the experiments, up to 16 processors was used.

Results The system was first tested with artificially generated input data. A separate processor was used to produce input data with the same distribution as data generated in the experiment. However, the random number generator used to produce the input data was unable to calculate more than about 3000 observations per second, so these test could not be carried out for input rates higher than this. The program was able to process these 3000 observations per second.

As the next step, the system was tested with real data from the experiment. In this case, the bottleneck proved to be the interface to the data files on the user host processor (in this case a Sun 3/160 workstation). Observations could be read from the disk at a maximum rate of about 1570 observations per second. The reason for this poor input/output performance was the primitive file store interface in the TDS programming environment.

Some experiments were carried out to measure the performance of the system without any input or output. These tests were carried out by processing the same input data a large number of times, so that data had to be input only once. These tests did not include the searching for clusters, but only the sorting of coordinates into the above described datastructure. The results from this test indicated that the system could handle over 4000 observations per second.

4.4 A multiprocessor system for full-text retrieval

The parallel full-text retrieval application was written by Marina Walldén (M.Sc. Thesis) and Kaisa Sere [Walldén and Sere 89] at Åbo Akademi.

Problem description The problem studied in this application was fast retrieval of information in large text databases. Consider a database containing a large amount of documents from different articles (e.g. from newspapers), law text, bibliographies etc. The user wishes to search the database by making queries in the form of search words which describe the topic the user is interested in. The system should report to the user the documents that contain the given search words.

Full-text databases can be very large and consequently a search for a given pattern can be very time-consuming. To solve this problem, a parallel implementation of a full-text retrieval system was constructed.

Database representation The database is distributed among a number of processors, so that each processor holds only a part of the database. Each document is stored as a whole in one processor. Documents are stored using a *surrogate coding* technique, which makes it possible to quickly determine whether a word occurs in a document or not.

A *surrogate table* is an array of k bits. For each search word that we wish to insert into the table, we calculate i hash codes using some suitable hash functions, each with a value between 0 and $k-1$. To store a word in the surrogate table, the bit-positions in the table given by the hash-codes are set to 1. The number of hash codes per word, i, is often between 10 and 30

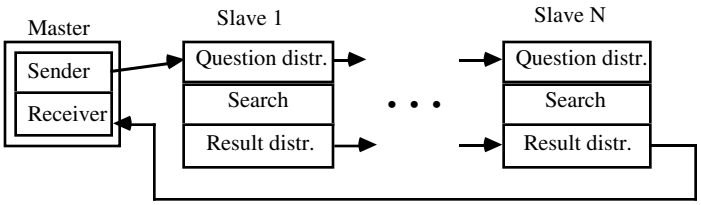

Figure 11: A ring processor farm

Processors	0.5 Mbyte	1 Mbyte	3 Mbyte	10 Mbyte
3	3.0	-	-	-
7	6.3	6.8	-	-
15	11.1	12.7	14.0	14.2
31	9.7	11.4	13.0	13.7
63	5.3	5.4	6.0	6.1

Table 2: Observed speed-up for ring structure

and k, the length of the surrogate tables is often 512 or 1024 bits. When we search for a word, the i hash-codes are calculated from each search word and the resulting bit-pattern is compared with the stored surrogate tables. If the corresponding bit positions in the surrogate table contain 1-bits, the word has been found in the document.

Parallel implementation The full-text retrieval system was implemented as a processor farm, with one master processor broadcasting queries to a number of identical slaves which search their own partition of the databas for the specified search words.

The master processor consists of two independent processes: a *sender* process which broadcasts search words to the slaves and a *receiver* process that accepts results from the slaves. A slave processor consists of three processes: a question distributer process which takes care of incoming search words, a search process which searches the processors part of the database for the given search words and a result distributor that sends the results from the search back to the master. The question distributor process contains buffers for search words so that the search process does not have to wait for communication, but can continue with the next search as soon as the previos has been completed.

The processor farm has been implemented on two different processor interconnection structures. The processor farm mapped to a unidirectional ring of processors is illustrated in Figure 11. The ring contains $N + 1$ processors of which one acts as a master and the other N act as slaves. Tests were carried out with 3, 7, 15, 31, and 63 processors. The size of the database was 0.5, 1, 3 and 10 Mbytes. The speed-up factors from the tests are summarized in Table 2. A dash (-) in the table means that the corresponding test case could not be implemented due to shortage of memory on the slave transputers.

The processor farm mapped to a binary tree is illustrated in Figure 12. The same tests were carried out on the tree structure as for the ring structure. The measured speed-up factors from the tests are presented in Table 3.

Conclusions From Table 2 and Table 3 we can see that configurations with up to 15 processors give a linear speed-up, which is very close to the number of processors used. For tests with 31 or

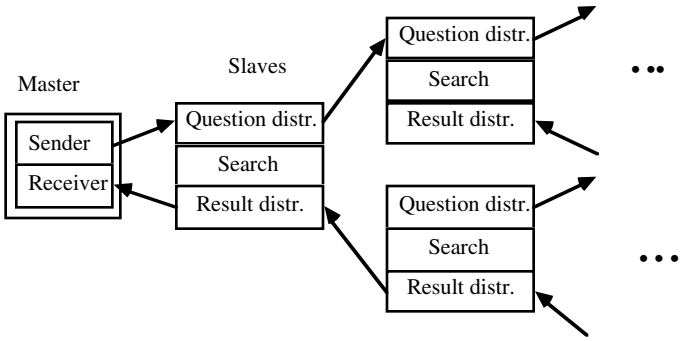

Figure 12: A tree processor farm

Processors	0.5 Mbyte	1 Mbyte	3 Mbyte	10 Mbyte
3	3.0	-	-	-
7	6.6	6.8	-	-
15	11.9	13.5	14.9	14.9
31	9.6	11.3	13.0	13.7
63	5.4	6.2	6.9	7.3

Table 3: Observed speed-up for tree structure

more processors, the speed-up decreases. The reason for this is that the portion of the database that is allocated to one processor becomes too small and thus the ratio between calculation and communication also becomes too small. However, the limited amount of memory available prevented tests with larger databases to be carried out. We can also see that slightly better results can be observed for the binary tree. The reason for this is that the average length of the communication path is much shorter in a tree than in a ring.

5 Conclusions and future work

The implemented applications show that a large class of computational problems can be solved efficiently using multiprocessor systems. However, a necessary condition for an efficient parallel implementation is that the problem is big enough, otherwise the effort of decomposing the problem between a number of parallel processors is not worth while. A good treatment of these problems can be found in [Fox et al. 88].

The applications where the problem can be decomposed into a number of relatively independent subproblems can be implemented very efficiently using the processor farm approach. If the processor farm approach can be used, the programming task is simplified by the fact that each slave processor executes a sequential algorithm solving a part of the original problem. As the communication structure is similar for all processor farms, it is possible to write program skeletons into which the user only needs to fill in the sequential code for the slave processors and the code for decomposing the original problem into independent subproblems. Using this approach, parallel programs can be constructed with a minimal amount of work, reusing existing sequential code. The processor farm has also proved to give very efficient parallel programs for sufficently large problems, as the system is automatically load balanced.

Problems where the amount of data that has to be processed is very large can often be geometrically decomposed by dividing the data domain evenly among the processors. In applications of this type, a processor often has to communicate with all neighbour processors. Existing sequential code can be reused also in this case, but the code has to be modified to take into consideration the fact that each processor only holds a part of the data and has to communicate with other processors if it needs values that reside outside its own domain. The amount of communication between two neighbour processors A and B is often proportional to the length of the border line dividing the data domain of processor A from the data domain of processor B. Geometrical parallelism is a natural way of decomposition for a large class of problems and has been showed to give good results. However, the amount of programming needed for this type of solutions can be quite large, especially in problems which involve decomposition of three-dimensional models.

The transputer technology has prooved to be very reliable. The Hathi-2 system has been in use since April 1988, and only one minor hardware error has occured during this time. One reason for this is the simple design of the Hathi-2 boards and the relatively small number of components used. The system has also prooved to be very flexible due to its reconfigurable switching network and easily accessed via the national data networks.

A perhaps somewhat unexpected observation that has been made during this work is that paralell programming in itself is not much more difficult than traditional sequential programming. An experienced programmer with a good knowledge about the application in question can easily decompose the problem and implement a parallel solution. The problems encountered in designing parallel programs have been caused by the lack of good programming tools, like debugging tools, automatic message routing facilities, monitoring tools etc. Further work on multiprocessing in the Department of Computer Science at Åbo Akademi is concentrated on these problems.

The Hathi project was followed up by the research program FINSOFT III: Parallel computation and neural networks, which is a subprogram of the FINSOFT research program financed by TEKES. The work at Åbo Akademi has been continued in the FINSOFT III program in two projects, Millipede and Centipede.

The goal of the Millipede project is to build an integrated programming environment for the construction of parallel programs [Aspnäs and Back 89]. A parallel program is represented graphically in the environment as a *process graph*. The processes can be grouped together into *tasks*, which are units that can be executed by a processor and which consist of one or more parallel processes. A *task graph* can be placed onto the processors in Hathi-2 by an automatic mapping tool, which allocates tasks to processors and logical communication channels between tasks to physical links. In this way, the physical structure of the multiprocessor system is hidden from the programmer, which only has to operate on a simple model of parallel processes which communicate via logical communication channels.

Acknowledgements

The Hathi-2 multiprocessor system was designed and built in the Hathi project, which was financed by the Technology Development Center (TEKES), The Academy of Finland, Åbo Akademi and the Technical Research Center of Finland (VTT). Part of the work has been done in the FINSOFT III research program, financed by TEKES. The authors wish to thank everybody that have been involved in the design of the Hathi system and its software. Tom Lönnroth, Aarne Rantala and Göran Öhman are gratefully acknowledged for helpful comments on the paper.

References

[**Aspnäs and Back 89**] M. Aspnäs and R.J.R. Back, A Programming Environment for a Transputer-Based Multiprocessor System, To appear in *Proc. First Finnish-Hungarian Workshop on Programming Languages and Software Tools*, Szeged, Hungary, 7-11.8.1989. (Also published in Reports on Computer Science & Mathematics, Åbo Akademi, Ser. A, No 82, 1989).

[**Aspnäs et al. 89**] M. Aspnäs, R.J.R. Back, T-E. Malén, The Hathi-2 Multiprocessor System, *Reports on Computer Science, Ser. A, No. 80*, Åbo Akademi, 1989.

[**Aspnäs and Malén 89**] M. Aspnäs and T-E. Malén, Hathi-2 users guide, *Reports on Computer Science, Ser. B, No. 6*, Åbo Akademi, 1989.

[**Burns et al. 88**] G. Burns et al., Trillium Operating System, *Proc. Third Conference on Hypercube Concurrent Computers and Applications*, ACM, 1988, pp. 374-376.

[**Fox et al. 88**] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon and D. Walker, *Solving Problems on Concurrent Processors, volume 1, General Techniques and Regular Problems*, Prentice-Hall, 1988.

[**Hoare 78**] C.A.R. Hoare, Communicating Sequential Processes, *Communications of the ACM, 21, 8 (Aug. 1978)*, pp. 666-677.

[**Inmos 88a**] Inmos Limited, *Transputer Reference Manual*, Prentice–Hall, 1988.

[**Inmos 88b**] Inmos Limited, *occam 2 Reference Manual*, Prentice–Hall, 1988.

[**Jones and Goldsmith 88**] G. Jones and M. Goldsmith, *Programming in occam 2*, Prentice–Hall, 1988.

[**Kilpinen et al. 89**] A. Kilpinen, T. Björkholm, T. Westerlund, Parallel Calculation of a Packed Bed Reactor, in *Modelling, Identification and Control, Proc. of the 8th IASTED International Conference*, Grindelwald, Switzerland, 1989.

[**Pehkonen 89**] K. Pehkonen, *A dynamically reconfigurable parallel computer Hathi-2*, Licentiate thesis, University of Oulu, Department of Electrical Engineering, 1989.

[**Perihelion 89**] Perihelion Software Limited, *The Helios Operating System*, Prentice-Hall, 1989.

[**Rantala et al. 88**] A. Rantala, A. Raunio and D-J. Still, A Multiprocessor System for Fast Geometric Image Transformation, *Reports on Computer Science, Ser. A, No. 66*, Åbo Akademi, 1988.

[**Rantala et al. 89**] A. Rantala, A. Raunio and D-J. Still, Some Parallel Implementations of a Geometric Image Transformation, *Proc. SCIA 89, The 6th Scandinavian Conference on Image Analysis*, Oulu, Finland, 1989.

[**Walldén and Sere 89**] M. Walldén and K. Sere, Free-Text Retrieval on Transputer Networks, *Microprocessors and Microsystems, Vol. 13, No. 3*, April 1989, pp. 179-187.

[**Öhman et al. 88**] G.A. Öhman, T-E. Malén and P. Kuusela, Numerical Fluid Flow and Heat Transfer Calculations on Multiprocessor Systems, *Heat Engineering Laboratory report 88-3*, Department of Chemical Engineering, Åbo Akademi, 1988.

[**Äijänen 88**] T. Äijänen, Distributed Interconnection of a Reconfigurable Multicomputer System, *Microprocessing and Microprogramming, 3-1988*, pp. 243-246.