Parallel computing within the OOTI programme

drs.ing. Ronald Reinds

Parallel computing is one of the central points in the OOTI programme. Students take obligatory classes on this topic and many do projects in this area, for example related to the EUT transputer network. This article gives an overview of the field of parallel computing and gives some examples of work done within the scope of OOTI.

In parallel computing, the aim is, in many cases, to increase the performance (compared to traditional single processor computers) by deploying more than one processor. In order to fully exploit this potential performance, often explicit parallel programs are needed. In explicit parallel programming, the programmer splits up the program into parts that operate concurrently. Important topics in parallel programming are

- how to balance the load of the parts to be executed evenly over the available processors, and
- how to arrange communication and synchronization between the parts of a parallel program.

It may be apparent that parallel programs introduce extra complications and offer extra possibilities for design decisions (in comparison with conventional programming). That is the main reason why a course in parallel programming is part of the OOTI programme. This article covers the main topics addressed in this course in a global manner.

Parallel machines

For getting the most out of a parallel program, one needs a parallel computer. Most parallel machines are based on the classical Von Neumann concept, i.e., every processor performs the following cycle continuously.

1. Instruction fetch (get instruction from memory to processor + decode the instruction),
2. Instruction execution.

By Flynn's taxonomy, these Von Neumann-based machines can be classified according to two criteria.

- Whether a machine can execute a single instruction only or multiple instructions at a given instant, and
- Whether it can operate on a single datum only or on multiple data.

This yields four classes.

1. Single Instruction, Single Data (SISD). To this class belong the traditional computers, like PCs and workstations.
2. Single Instruction, Multiple Data (SIMD). Computers in this class consist of a large number of small processors. Each processor executes the same program, but on different data items. An examples of a computer belonging to this class is: the Connection Machine (the older ones like CM2).
3. Multiple Instruction, Single Data (MISD). Empty class.
4. Multiple Instruction, Multiple Data (MIMD). In this class, the most parallel machines are found. Examples are: transputer networks and Ncubes.

In 1990, our university acquired a transputer network as a platform for parallel computing. This network is a MIMD machine as mentioned above. One reason for the interest in MIMD machines is the possibility to build a powerful machine by interconnecting a large number of cheap and simple processors. The price/performance ratio is, in many cases, better than traditional single processor machines having the same speed. For parallel systems, there are two notions that are important for quantifying the performance, viz. *latency* and *throughput*. Latency is the time that elapses from the instant a problem has been given to the system until the moment the answer comes out. Throughput is the rate with which the system is capable of communicating with its environment reading data

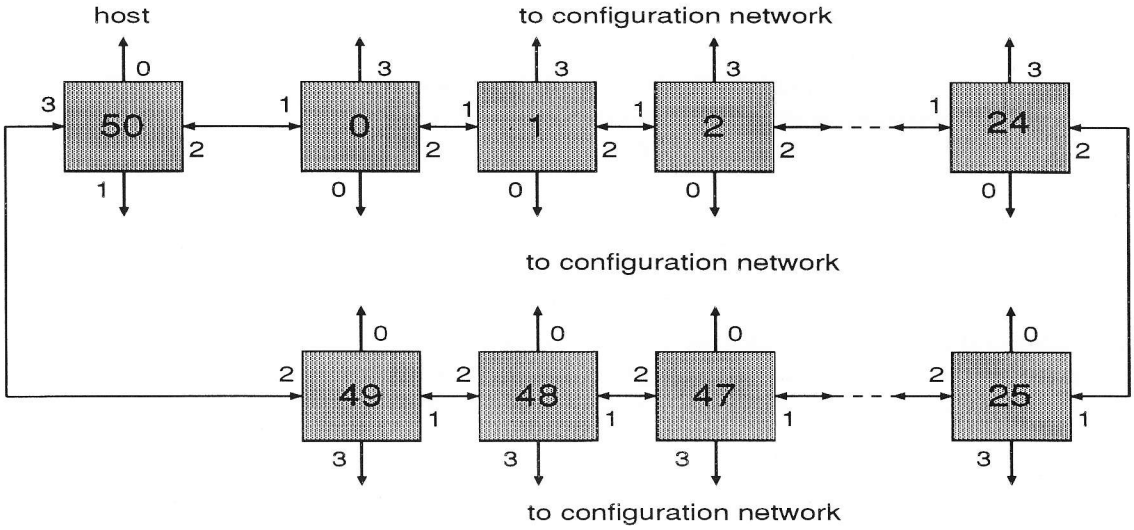

Figure 1: The hardwired configuration of the transputer network

and producing answers. It is apparent that the latency is bounded by the throughput.

The network owned by the university consists of fifty transputers distributed over four boards in a single cabinet. A transputer is a chip containing a processor, some memory, and four bidirectional links for connection to other transputers. Some links are hardwired, other ones can be set by software. The network does not operate stand-alone, it is controlled by a transputer that resides in a host machine, typically a PC or a workstation. The transputer that is connected to the host is called the *root* (of the network). The network is depicted in Figure 1. In this figure, the hardwired links are the ones in between transputer 50 (the root) and transputer 1, and the links between transputer i and transputer $i+1$ ($0 < i < 50$). The soft-links are the ones connected to the configuration network; this network is programmable.

Software

There are three ways to introduce parallelism in software.

1. Recognition of parallelism by a compiler,
2. By choosing a language that allows a parallel implementation,
3. By designing a new language that allows a parallel implementation.

Recognition of parallelism by a compiler is often very hard, since it is very difficult to detect parallelism when it is not specified. Furthermore, when the program is not regular, it is hard to get a good load balance over the available processors. But the main problem is that a good parallel program for many problems differs from a sequential solution and that it can not be derived in an automatic fashion from a sequential program or algorithm. So, in many cases, it is better that the programmer explicitly splits up the program in parts that operate concurrently; these parts are often called *processes*. However, in explicit parallel programming new problems pop up.

- How to deal with communication and synchronization between the parts of a parallel program.
- How to prevent *deadlock*.

These problems are related in some way. For explicit parallel programming, there is often a need for communication and synchronization between the processes. Communication and synchronization can only take place when all (in many cases only two) processes involved are ready for it. Processes can only become suspended on a communication or a synchronization action. When the partner process in communication or synchronization is not ready, the process that is willing to communicate or to synchronize gets suspended. This is because the communication is synchronous, i.e., a message is transmitted only when it will be accepted by the other party. In a deadlock situation, all processes got stuck, because they are all waiting on a communication or synchronization action to be performed by another process. In the following paragraphs, we will address these problems.

How to make programs for a network

Processes can communicate using *channels*. A channel is a one-directional process-to-process connection. One process can only read from it, its partner process can only write on it. A parallel program consists of a number of processes and channels connecting processes. A parallel network contains processors and (physical) links interconnecting the processors. So both, a parallel program and a parallel network can be viewed as graphs. It seems natural to map processes in the program onto processors in the network and channels onto the links. When a program is realized as a VLSI circuit, this may be the way to do it. However, if the program is to be executed by a network, it may not be possible to do the mapping mentioned above, because the number of processes of the program may exceed the number of processors in the network or the channels interconnecting the processes can not be mapped one-to-one to physical links. A way to overcome the first problem is to run more than one process on a single processor using *time-slicing*, i.e., the processor will be allocated to a process for a period of time (called time slice), then the process is pre-empted and a new process is scheduled when the time slice expires; the process will be re-scheduled to continue execution at a later time. The second problem can be solved by mapping one channel onto a number of links (*routing*) in combination with the mapping of several channels onto a single link (multiplexing). In general, it is a good philosophy to distinguish the design of a program and the mapping onto a parallel network. The mapping can be performed more or less mechanically. One step that can be done independent of the program is the construction of a routing mechanism. Such a mechanism is added to the network and implements a connection between every pair of processes (provided that the network is strongly connected). This mechanism can be implemented both in hardware and software. The network plus the routing mechanism can be considered as a fully connected network. In the course, some efficient algorithms that perform multiplexing and routing are discussed.

Our programs consists of a collection of communicating processes. For reasoning about parallel programs, there are two important properties, viz. *safety* and *progress*. The safety property prescribes that nothing goes wrong. The progress property states that eventually something must happen. This forbids a trivial implementation, viz. one that does nothing. In the case of a sequential program, we are only interested in progress towards the end of the program. In a process, there are often a lot of places where it may become suspended. For each of these places or states, there is a potential risk that the process remains suspended forever. A state in which a process remains suspended unless its environment interacts with it is called a *stable state*. So, when designing a parallel program, we have to prove the absence of deadlock, i.e., the absence of unwanted stable states. Proving absence of deadlock can be done by (formal or less formal) reasoning, although it is often quite laborious.

How to program the transputer network

For programming the transputer network, we use Transputer Pascal. This is a sort of extension to standard Pascal. However, the language does not support files. A program written in standard Pascal consists of a declaration part and a main program. This main program can be viewed as a process and, therefore, a Pascal program is in fact a single process program. In order to support parallelism, some features have been added. Since it is very easy and cheap to have several processes running on a single transputer, the Transputer Pascal compiler supports parallelism at statement level and at procedure level as well. When a compound statement is enclosed by **cobegin** and **coend** (just like **begin** and **end** in standard Pascal) the statements in this compound statement are executed concurrently. If a procedure call is preceded with the keyword **fork**, this call is executed in parallel with its caller. Transputer Pascal also has some other extensions, but we do not discuss them here. The Transputer Pascal compiler compiles a single Pascal program into code for a single transputer. Every transputer in the network must have its own program. It is already mentioned that the transputer connected to the host is the root. It is natural to use this transputer for communication between the host and the network. The root runs often another program than the network transputers. There should be a facility for routing the programs through the network towards its destination. The program that takes care of this routing is called *tmon*. As input for this program, the programmer has to specify for any transputer which program it has to run (plus some other information). The tmon program performs, among other things, the following tasks.

- call the compiler for the programs,
- contact the host machine of the transputer network,
- load the programs, and
- remain as monitor.

The tmon program provides also facilities to maintain an X-window (e.g., for drawing pictures).

Examples

We conclude with two examples that illustrate two ways of introducing parallelism in a program, viz. *processor farming* and *data distribution*. Last year, the examples discussed below made up the practical part of the course. In both examples, the transputers in the network will run two processes, viz. one for communication to the exterior and routing, and one that performs the calculation of the results (see Figure 2).

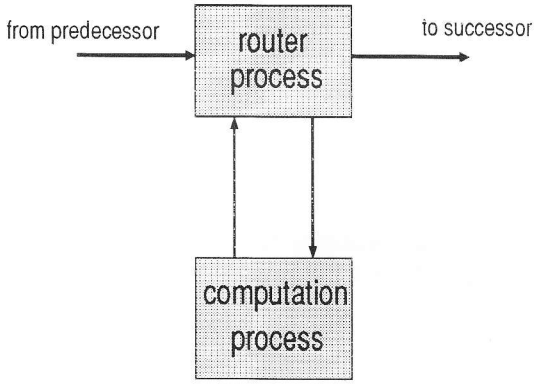

Figure 2: Processes running on a single network processor

Drawing fractals

A algorithm that is easy to *parallelize* is the calculation of fractals. A window on a computer screen can considered to be a discrete complex plane within certain boundaries. So, each pixel on the screen has a complex number, say c, associated with it. For getting Mandelbrot images, we repeatedly apply $f(n+1) = f(n)^2 = c$ (starting with $f(1) = 0$), where $f(i)$) is a complex number for every natural number i and c is a non-zero complex value, until the modulus is equal to 2 or exceeds 2. The number of iterations (below a certain maximum) before the series starts to diverge, determines the colour of a pixel. It is apparent that the calculation of the number of iterations before the series starts to fall to infinity for a complex number is independent of any other number in the complex plane. A natural way to parallelize the calculation is to let every processor calculate the results for a part of the plane (e.g., one line); this is called a job. The router process determines dependent on the load of the computing process whether an incoming job can be done on its own processor or should be forwarded to another processor. In this way, we get a dynamic load balance. The transputer connected to the host runs two processes, viz. one for sending jobs to be executed by the transputer network and one for receiving and collecting results of the calculations. This is an example of *processor farming*. In processor farming, the domain of the total computation is split into parts and the computation of these parts are independent.

Gauss-Jordan elimination

A field that is very popular in parallel computing is matrix computation, because it is often easy to make parallel algorithm for this type of computation. An example is Gauss-Jordan elimination. This is a well-known technique to solve a set of linear equations represented as $Ax = b$. Here, matrix A is a non-singular, $N \times N$ matrix of coefficients, b is vector of length N, and x is the vector of unknowns to be solved. Gauss-Jordan elimination is based on the fact that the solution of the set of equations is not affected by any of the following operations.

1. Swap two rows of A and the two corresponding values in b.
2. Replace a row of A by a linear equation of itself and another row and perform the same operation on the corresponding values of vector b.

By repeatedly applying these elementary operations, matrix A can be transformed such that solving the set of equations becomes trivial. This can be done in two ways: transform A into a triangular matrix or transform it into the unity matrix. In the first case, we only have to solve a triangular system, which is easy. In the latter case, the value of b is the solution vector.

We give a brief outline of the parallel algorithm. First, the rows of matrix A and the corresponding values of b are distributed evenly over the available processors. In each step of the algorithm, one of

the columns of A is replaced by one of the unit vectors by applying elementary operations. The algorithm transforms the columns in increasing order. For the sake of convenience, let i be the current column. For transformation of column i, a pivot row must be chosen and must be distributed over all available processors. A pivot row should have a non-zero element at position i On each processor, the computation process will adjust the rows that are assigned to this processor. This should be done in a way that the elements in column i will form a unit vector. Rows other than the pivot row should be replaced by a linear combination of itself and the pivot row. The processor that hold the pivot row should normalize it, i.e., scale the elements in the pivot row such that the element at position i gets equal to 1. In this example, the data (matrix A and vector b) are distributed over the available processors. However, for adjusting the rows other than the pivot row, each processor must to have the pivot row. This way of introducing parallelism is called *domain decomposition* (or *data parallelism*). In data decomposition, we divide the data among the available processors in order to divide the computation steps evenly, but the computation of parts is no longer independent.

Additional reading

J.J. Lukkien, Parallel Program and Computer Networks (draft lecture notes).
J.J. Lukkien et al., The Eindhoven Transputer System, an overview.
J.J. Lukkien, On multiplexing, routing and Remote Process Calls.
J.J. Lukkien, Parallel Gauss-Jordan Elimination.
□

Drs.ing. Ronald Reinds is a student of the postmasters programme Software Technology. He is a member of XOOTIC.

Short News

Block instructors and Programme Group
Two new block instructors have been appointed. Prof.dr.ir. Koos Rooda (faculty of Mechanical Engineering) has succeeded prof.dr.ir. Ton van de Wolf as block instructor for *Discrete Manufacturing*. Prof.dr. Jos Baeten has taken the place of prof.dr. Kees van Hee as block instructor of *Formal Specification Methods*. Both new block instructors also have taken the place of their predecessors in the Programme Group (*Opleidingsgroep*).
In that same Programme Group more changes have taken place. Prof.dr.dipl.ing. Dieter Hammer has succeeded prof.dr. Martin Rem as chairman. Rem will stay as ordinary member. Dr. Anne Kaldewaij is added to the Programme Group and dr. Onno van Roosmalen is temporarily added as replacement of Hammer, who is on a sabbathical leave. □

Discrete Manufacturing revised
With the arrival of a new block instructor for *Discrete Manufacturing*, the contents of this block has been revised. The new contents is as follows.

- 4U020: Numerical control (Hijink)
- 4C340: Production technique (Mikkers, Rooda, Schellekens)
- 4C710: Machine control, processes, and interactions (Rooda, Van Rooij)
- 1K061: Re-design of technical production systems (Sanders)
- 1K110: Capita selecta Automation of technical production systems (Sol)
- 2J594: Small project Discrete Manufacturing (Struik or Rooda) □

More curriculum changes
The contents of the course block *Laboratory Automation* has been revised as well. During the academic year 1994/95 it will be called *Process Automation*. From 1995/96 on it be called *Embedded Systems*. Prof.dr.dipl.ing. Dieter Hammer will be responsible for the contents of that block.
Starting in the academic year 1995/96 OOTI will offer a new block to its students, called *Medical Applications*. Dr.ir. Kees van Overveld will as block instructor be responsible for the contents. XOOTIC MAGAZINE hopes to report on this new course block when more is known about the contents. □