PARALLEL ALGORITHM DEVELOPMENT FOR A SPECIFIC MACHINE (INMOS T800 TRANSPUTER)

BY

WILLIAM STAUB
This paper will describe from start to finish the parallel algorithm program development for a specific transputer (Inmos T800) network. It is essential to understand how the hardware manages parallel tasks and how information is exchanged between the transputers before writing a parallel program.

A transputer (figure 1, page 3) is a circuit containing a processor, some memory to store programs and data, and several ports for exchanging, or transferring information with other transputers or with the outside world. By designing these circuits so that they could be connected together with the same simplicity with which transistors can be in a computer, the transputer was born.

One of the most important factors was the introduction of a high-level language, occam [MAY83], whose features were directly supported by this transputer's hardware and that made the transputer a building block for parallel computers. However a Logical System C compiler was developed for wider known usage.

A prominent factor to utilizing this circuitry was the ease with which transputers could be connected to each other with as little as a few electrical wires. The four bi-directional input/output (I/O) ports of the transputer are designed to interface directly with the ports of other transputers, this feature allows for several transputers to fit on a small footprint, with very little extra logic circuits, making it possible to easily fit four transputers with some memory on a PC daughter board (ISA bus).
Figure 1 shows the basic block diagram of a transputer, along with some interconnection schemes: a linear chain (a), offering the simplest of connections, a ring (b), and a mesh (c).

The processor does not have dedicated registers, but a stack of registers, which allows for an implicit selection of the registers. The net result is a smaller instruction format which leads to a denser program code. The transputer adopts the RISC philosophy and supports a small set of instructions executed in a few cycles each.

Multitasking is supported in microcode within the processor. The actions necessary for the transputer to swap from one task to another are executed at the hardware level, freeing the system programmer of this task, and resulting in fast swap operations. This is an important way to improve the performance of a processor by allowing it to start running another program if the one that it is currently executing cannot continue for awhile. Multitasking is the first form of parallelism supported by the
transputer, and is accomplished by having the processor maintain a list of tasks that must be executed. Each task executes for a small amount of time, called a quantum, before it is stopped, and swapped for another task (if one awaits). For the transputer, a quantum of time is typically on the order of two milliseconds (2 \times 10^{-3} \text{ seconds})[1]. Tasks are executed in a round-robin fashion. When a task eventually terminates it is removed from the list. The transputer maintains the active tasks chained in a linked list, and two of its internal registers are used to point to the front and rear of the list. The actual list is stored in memory, and the registers contain the memory address of the cells defining the tasks. At any time, new tasks may be created and added to this list. A transputer task may be in either one of two states:

**Active:**

- This state refers to a task that is being executed, or in the list of tasks waiting to be executed.

**Inactive**

This state refers to a task that is not in the list of active tasks, because either one of three conditions is preventing it from continuing execution:

- The task is waiting for an input from one of the I/O ports
- the task is waiting to output to one of the I/O ports, or
- the task has been asked to stay idle until a specified time in the future.
As far as RAM, the transputer can access a linear address space of 4 Gbytes. Of these 4 Gbytes, 4 Kbytes are built-in the T805 transputer circuit, and correspond to the lower part of the memory address space. Since the registers are 32 bits in length, the transputer accesses 4 bytes at a time when accessing memory.

The combination of the architecture of the serial I/O ports and the way the transputer manages them contribute to making the transputer a unique circuit especially well tailored for multiprocessing. I/O ports are memory mapped. This means that programming a port and passing it the address and length of a message is done by writing these numbers to memory locations that are mapped to the registers of the port. The four bi-directional serial comprise two wires. When two transputers are connected together, they exchange information through one of their links. The processes (or tasks) running on each of the transputers are then free to communicate by exchanging data or messages over the link. The term link is used to refer to the physical connection between two transputers, and the term channel to is used to describe the software connection between the two processes. The transfer of data over a serial link is synchronized and unbuffered.

- **Synchronization** refers to the fact that if Process $P_1$ running on Transputer $T_1$ needs to exchange (send or receive) information with Process $P_2$ on Transputer $T_2$, then it must wait until $P_2$ is ready to participate in the exchange. The processes are said to be synchronized, since the sender cannot get rid of its data until the receiver is ready to accept them, and conversely.

- **Unbuffered communication** means that no temporary storage is needed to store incoming or outgoing messages. The transfer takes place directly between
the memory of the sender and the memory of the receiver. The I/O port is a
*Direct Memory Access Controller* (DMAC) which is given the memory address of
the first byte of the incoming or outgoing message, and the number of bytes
contained in that message. Once initialized, the link is autonomous, and will wait
for its partner to signal its readiness to participate in the communication.

During the communication, the processes that initiated the transfer are **blocked**. Each
process is placed at the rear of the list of inactive tasks. Because the processor and the
links operate independently, the processor is free to run another process when one is
blocked by communication. As a result the channels require no message queues or
message buffers.

Data are transferred one byte at a time, each byte generating an acknowledge
from the receiving transputer. The typical transfer speed is 10 Mbits/sec, although
speeds of 5 and 20 Mbits/sec are also supported. When two transputers exchange data
in both directions, the bandwidth on the link can reach a maximum of 2.35 Mbytes/sec.

A transputer system operates as an *attached processor* for the PC. As such, it
does not replace the 80x86 processor in the PC, but works in conjunction with it. The
PC is referred to as the *host machine*. None of the software running on the PC is
affected by the presence of the transputer network. Only software written by the user
with the Logical System Compiler will be able to make use of the multiprocessing
capability. Figure 2, page 7 illustrates the general architecture of a PC with an attached
transputer network.
One of the transputers in the network is a designated driver (or root) and connects to the PC in one of the 8-bit ISA I/O slots. The other transputers in the network only require a power source (+5V) and thus need not be connected to the system bus. There, the root transputer it dispatches transferring information from the PC to the transputer network (program and input data), or from the transputer network to the host (result data). Unlike the other transputers in the network which use their links to connect to other transputers, the root transputer dedicates one of its links (often Link 0) to communicate with the PC. This link is connected to an 8-bit I/O controller serving as an interface between the PC and its resources, and the transputer network.

*Figure 2: The transputer as an attached-processor.*
This I/O controller is quite simple and consists of two 8-bit buffers, one for data input, the other for data output. Each buffer is associated with a 1-bit status indicating whether the buffer it represents is full or empty. The 80x86 host thus sees the interface as four 8-bit registers. The root transputer sees the port as another transputer with which it exchanges data serially.

The I/O controller manages the parallel to serial conversion (and conversely) between the PC and the transputer. Consequently, the transputer cannot directly control the resources of the host systems: user memory, file system, keyboard, or video display. Every access to the PC resources must go through the 8-bit I/O port, which is maintained, in software, by the 80x86. If the transputer needs to display a string of characters on the video screen, for example, it must send the 80x86 the string, along with a command telling the 80x86 what to do with it. The Logical Systems C compiler allows us to program the root transputer to execute a printf() statement, this statement cannot be carried out unless the 80x86 is running a program that can carry out the operation. For this reason the execution of parallel programs on the transputer networks will always involve the 80x86 host in some respect.

- Sometimes the host will become a simple subordinate of the root transputer, attending to its every needs and making all its resources available to the transputer: keyboard, disk, video display, speaker, etc. Logical Systems provides a special program when this mode of computation is required called CIO.
- At other times, though, it will be more advantageous to divide the computation between the transputer network and the 80x86, and leave all I/O related operations to the latter. This will result in a better utilization of the system.
Remember that running a program on the transputer requires running two programs: One on the host, and the other on the transputer network. With the help of batch files, the automation will make writing and running programs a smooth task. The software is an ANSI C compiler augmented with libraries written by Logical Systems to support parallel constructs. The package contains the standard preprocessor, compiler, assembler and linker programs that most C programmers will be familiar with, but also new utilities, such as a loader and a host driver, required by the parallel environment.

The process (figure 3 below) takes the original file first.c and passes it through a preprocessor (1), a compiler (2), an assembler (3), a linker (4), and a loader (5), this last one loads the code in the transputer network and starts its execution.

![Figure 3 The Compilation Process](image-url)
Loading the transputer program on the transputer network, and launching it requires ld-net to perform several tasks:

- It must load the program into the root transputer.
- Once the parallel program is loaded, the loader must give the signal to the root transputer to start execution.
- It must start the execution of a program on the PC host computer so that input and output requests by the root transputer can be carried out.

**Network information file**

The loader gets the information about how it should perform these three steps from a file, the *network information file*, which typically carries a .nif extension. The contents of the network information file consists of two parts. The first section contains four lines controlling the loading process. The second section contains information describing the graph representing the transputer network. **This information is used by ld-net to determine which program or programs must be loaded onto the transputers.** An example network information file for our program is shown in the listing below.

```
buffer_size 200;
host_server cio.exe;
level_timeout 1000;
decode_timeout 1000;
1, first, R0, 0, ,,;
```

Listing 3-5: contents of nif file first.nif for program first.c.
Commands
The first four lines of the file form the command part of the network information file.

Buffer_size
Buffer_size defines the number of bytes of transputer internal memory used when downloading programs. When a program running on several transputers is loaded, it is first passed to the root-transputer, which passes it on to neighboring transputers, which, in turn, pass the program down to their neighbors, and so on. A buffer is used to hold the data transferred. The default value is 255 bytes, and results in the fastest loading time possible. Smaller buffer sizes are possible, but can substantially slow down the loading of programs in large transputer networks.

Host_server
Host_server identifies the program that is to run on the PC host. We will first use the cio.exe program provided by LSC.

Level_timeout
To understand how level_timeout and decode_timeout (see below) operate, one has to understand the way a transputer network is loaded. The nodes are organized as a tree, with the root transputer coinciding with the root of the tree. Each node is loaded by its parent, and is responsible for passing on modules to be loaded on its child(ren). The level_timeout quantity represents the number of milliseconds required for a node to send a message (program to be loaded) down to its child nodes, and to receive an acknowledge of successful execution. The range is 25 to 1000 milliseconds. If the root transputer or any other transputer in the network does not receive a "transputer loaded" acknowledge from all its children during the allotted time, a time-out condition occurs and the loader stops.

decode_timeout
The decode_timeout quantity is similar to the level_timeout, except that it defines the maximum amount of time required by a single node for getting a message and executing it. An order to clear the memory, for example, is one that requires a nonnegligible amount of time. The default value is 1000 (or one second), but the allowed range is from 25 to 20000. Most programs running on average size networks will load without problems using the default values. Larger programs, or complex networks may require experimenting with different values.

Node Description
The second part of the network information file simply describes the physical configuration of the transputer network, including neighbor-to-neighbor links, and the allocation of programs to transputers. The format for each line is the following:
Node#, Program, Parent, [Link0], [Link1], [Link2], [Link3];
which, in the context of our program first.c, results in:
1, first, R0, 0, , ,;
**Node#**

Because first.c is written for one transputer only, our network has only one node, and therefore one line is sufficient to describe it[5]. Its first field, 1, indicates that we are using the root transputer. All transputers are assigned a different Id, with the root transputer always assigned Id 1.

**Program**

First indicates that the program to be loaded is first (the .tld extension is implicit). The parent field R0 indicates that the parent of this transputer is Node 0 (the PC host) and that the *parent* will reset this transputer through the master reset signal. This makes better sense when the network is a tree, for example, with the root transputer at the root of the tree. Then every transputer node has a parent. The parent of the root node is the PC host, which is always given Id 0. The R symbol indicates how the parent resets the current node. The letter R is used when the reset is passed from Parent to Child through the normal reset channel. The letter S indicates that the reset signal is generated by the parent, rather than passed from an ancestor.

Finally, the remainder of the line, 0, , , ; defines how Transputer 1 connects to the rest of the network through its links. Here, Link0 is connected to the host, with Id 0. The other links are not connected to any transputer involved in the parallel program, and are left blank.

To run the program, the command `<ld-net “first.tld” > reads the file first.nif and loads the executables within the transputer network and loads cio.exe within the host and starts execution.

Here is a source file (`prime_v1.c`, page13) for finding prime numbers that forms a linear chain network topology. This will show transputer-to-transputer communication, examining how programs running on neighboring transputers exchange information. But first we need some link hardware labels (Figure 4, page13).
The prime_v1 algorithm below works as follows. To find if integer x is prime, the program attempts to divide x by all integers lower than $\sqrt{x}$. Testing up to $\sqrt{x}$ is sufficient, but requires a square root or a multiplication operation. Our current goal is not to compute primes efficiently, but to explore different ways to parallelize the program with Logical Systems' parallel library.

```c
/*
 * prime_v1.c
DESCRIPTION: Computes primes on 2 nodes.

Node 2 computes the primes and sends them to Node 1. Node 1 (Root) collects the numbers and prints them.

TO COMPILE AND RUN:

make -f prime_v1
chainnif -# 2 -1 prime_v1 -nif prime_v1.nif

ASSOCIATED NIF-FILE
buffer_size     200;
*/
```
host_server  CIO.EXE;
level_timeout  400;
decode_timeout 2000;

1, prime_v1, R0, 0, 2, ,;
2, prime_v1, R1, 1, , ,;
======================================================================
#include <stdio.h>
#include <stdlib.h>
#include <conc.h>  /* transputer library */

/* ===============================================================
/* NODE #1 
/* ===============================================================*/
/*--- Receive information and prints it ---*/
if (_node_number==1)
{
    /*--- while numbers are coming, get them and print them---*/
    do
    {
        /*--- get input-channel address ---*/
        ChanIn(LINK1IN, &x, INTSIZE);
        if (x!=SENTINEL)
        {
            printf("%8d", x);
            NoPrimes++;
        }
    }
    while (x!=SENTINEL);
    printf("\nReceived %d primes \n", NoPrimes);
    exit(0);
}
else
/* ===============================================================
/* NODE #2 
/* ==============================================================*/
{ /*--- scan interval ---*/
    for (x = 1; x<INTERVAL; x++)
{ /*--- if prime then send number to Node 1 ---*/
    if (IsPrime(x))
        ChanOut(LINK0OUT, &x, INTSIZE);
}

/*--- signal Node 1 that we are done ---*/
    x = SENTINEL;
    ChanOut(LINK0OUT, &x, INTSIZE);
}


/* ISPRIME */
/* Semi-efficient prime finding function which, given x, returns 1 if*/
/* it is prime, and 0 otherwise*/
/* *--------------------------------------------------------------------*/
int IsPrime(int x)
{
    int i;
    static int SmallPrimes[10] = {0,0,1,1,0,1,0,1,0,0};
    if (x<10) return SmallPrimes[x];
    if (x%2==0) return 0;
    if (x%3==0) return 0;
    if (x%5==0) return 0;
    for (i = 2; i*i<=x; i++)
        if (x%i==0) return 0;
    return 1;
}

The output of the program is shown below.

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

Received 25 primes

The first section contains the code that runs on the root transputer (Transputer 1), while
the second one contains the code running on Transputer 2. The same program is
loaded on both transputers, but a different section of the code is actually used by each.

**ChanIn()**

The ChanIn function stands for Channel-Input. It receives information from a channel,
here LINK1IN (a constant predefined in conc.h), and stores it at the address specified
by the type-cast pointer &x. LINK1IN is the memory address of the I/O Port
converting to the input side of the link. We saw in Chapter 2 that the I/O ports are
memory-mapped. LINK1IN is thus a macro representing the memory address of the
port. INTSIZE is a macro that is defined at the beginning of the program as representing the size of an integer. Hence the function of ChanIn is to receive some amount of information from a channel and to store that information at a location defined by a pointer. The size of the information received is defined by an integer representing the number of bytes contained in the message. The prototype for ChanIn is: \texttt{void ChanIn(Channel *, void *, int)}

\textbf{ChanOut()}

ChanOut stands for \texttt{Channel Output}. It is the companion of the ChanIn function executed by Node 1. Its syntax is exactly the same as that of ChanIn:
\texttt{void ChanOut(Channel *, void *, int)}

It needs a channel pointer indicating which channel the information is sent over. In our case, the predefined constant \texttt{LINK0OUT} is used. It corresponds to \texttt{LINK0}, and specifies the outgoing direction. This information is then followed by a pointer to the area of memory containing the information to send. Here again a void pointer identifies the memory address of the information to be sent, and the third argument specifies how many bytes constitute the message.

\textbf{Blocking}

Blocking means that no transfer can take place unless both tasks exchanging data are ready to do so. In the context of our example, this means that if Node 2 is the first one to reach the \texttt{ChanOut(LINK0OUT,...)} statement, then it will not send any information until Node 1 executes the matching ChanIn(LINK1IN,...) statement. Moreover, the task running on Node 2 goes to sleep as soon as it executes the ChanOut statement, and does not awake until the transfer has completed. This was implemented by Inmos to allow other tasks to run on a transputer while one is awaiting a transfer through a channel. Figure 5, page 17 shows timing of a ChanOut (NODE2) operation occurring before its matching ChanIn (NODE1, or ROOT).
Figure 4.1: Timing of a ChanOut operation occurring before its matching ChanIn.
CONCLUSION:

The transputer is a very flexible and cost effective way of implementing a powerful multicomputer network.

It is flexible in that the two wire high speed bi-directional links allow easy configurations of various network topologies such as linear chains, rings, meshes, and hyper cubes. It being available as an ISA of PCI card allows easy interface to a PC compatible system bus.

The cost effectiveness stems from the fact that the transputer is a very low cost isolated computer. It’s usage allows mastering multiple computer concepts with only a limited expenditure.
Bibliography:

