A TOPOLOGY INDEPENDENT PARALLEL DEVELOPMENT
ENVIRONMENT

By
Norman J. Goldstein
B. Sc. (Mathematics) McGill University
Ph. D. (Mathematics) Cornell University

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

in
THE FACULTY OF GRADUATE STUDIES
COMPUTER SCIENCE

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA
April 1991
© Norman J. Goldstein, 1991
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
Vancouver, Canada

Date 18 Sept 1991
Abstract

This thesis describes a topology independent parallel programming environment, along with the motivation for its inception and design. The system is currently being used on a 74 node reconfigurable transputer network. A user’s program is described to the system using a high level of abstraction. This speeds up program development and facilitates the modifying of program parameters such as “problem size”. The system provides intelligent choices as the user’s program is bound to the hardware. The binding may be controlled to any extent desired by the user. This allows the hardware to be used efficiently, while relieving the user from having to know intimate details of the underlying hardware architecture.
# Table of Contents

Abstract

List of Tables

List of Figures

Acknowledgement

1 Introduction

1.1 Description

1.2 Other Work

2 Hardware and Maintenance Software

2.1 Hardware

2.2 System Software

2.2.1 Resources

2.2.2 Hardware Database

2.2.3 Ports

2.2.4 Crossbars

2.2.5 Session Setup

3 Crossbar Control

3.1 Introduction

3.2 Description of the algorithm
3.3 Derivation of the algorithm ............................................. 20

4 System Design ........................................................................ 22
  4.1 The Front end .................................................................. 22
  4.2 User Input ..................................................................... 23
  4.3 The Back end .................................................................. 25
  4.4 Outline ......................................................................... 27

5 Sample Session ....................................................................... 29

6 Channels .................................................................................. 33
  6.1 Events, Virtual circuits and ID's ...................................... 33
  6.2 Local Channels ................................................................ 35
  6.3 Non-local Channels .......................................................... 36
  6.4 Channel Properties .......................................................... 37

7 Resource & Place Modules ..................................................... 39
  7.1 Resource module .............................................................. 39
    7.1.1 Problem formulation .................................................. 40
    7.1.2 Constraints ............................................................... 40
    7.1.3 Cost function ........................................................... 41
    7.1.4 The constants ........................................................... 42
  7.2 Place module ................................................................. 43
    7.2.1 Outline ................................................................. 44
    7.2.2 Discussion .............................................................. 45

8 Setup & Load Modules .......................................................... 46
  8.1 Setup module ................................................................. 46
C.1.3 LoadLst ......................................................... 84
C.1.4 ChanVIds ......................................................... 85
C.1.5 Resources ......................................................... 86
C.1.6 LkData ............................................................ 89
C.1.7 ChanLks ........................................................... 89
C.2 Prep-p files ......................................................... 90
  C.2.1 ProcLst ......................................................... 90
  C.2.2 IOAdjLst ......................................................... 91
  C.2.3 CodeSize ....................................................... 92
  C.2.4 Degrees ....................................................... 92
  C.2.5 EdgeWts ....................................................... 92
  C.2.6 VipToRp ....................................................... 93

D Sample Source Files ........................................... 94
  D.1 Control Information ........................................... 94
  D.2 Host Code ....................................................... 95
  D.3 Root Code ....................................................... 97
  D.4 Node Code ....................................................... 98
  D.5 Substitutions .................................................. 99
List of Tables

2.1 Ports to access the network ........................................ 11
2.2 Reset Components .................................................. 11
7.3 For the constant $K_1$ of the cost function ...................... 43
9.4 Spring example ITB times .......................................... 49
9.5 Spring example program times .................................... 49
9.6 Ring example program times ....................................... 50
List of Figures

3.1 The crossbar control chain. ........................................ 18
3.2 The 3 switch setting procedures. .................................. 19
5.3 The graph of vnodes and channels. ................................. 31
Acknowledgement

I take this opportunity to thank the many people who have helped me put this thesis together. Their support covers the broad range of moral, technical and financial. You know who you are.

The thesis topic was suggested to me by my thesis advisor Alan Wagner who offered his own insights and ideas throughout the whole process. I wanted a problem with a large implementation component and I got it. I am grateful that my work is forming a basis for further projects in this area at UBC. Alan’s organizational skills and planning are responsible for all this coming together.

My parents Henry and Ethel were always there for me. For the last year, I have been fortunate to have the friendship of Rivian Rimer. This made the long work hours more bearable and productive; and it looks good that the relationship will continue for a lot longer.

The 4 people with whom I used the transputer hardware the most are Jie Jiang, Hilde Larsen, Ola Siksik and H. Sreekantatswamy. It was a pleasure to learn together and share with each other. A special thanks to Jie for imparting his hard-earned understanding of our transputer system.

From my first days at the department, our Head, Maria Klawe, was appreciative of my being there and supportive of my efforts. To her a hearty “Thank you”.

Norm Hutchinson made several helpful suggestions, and the time he put into reading this thesis is much appreciated.

There are people who made the non-technical (and some technical) sides of life more pleasant – eating, hiking, canoeing, singing and just plain hanging out. They include Yoko Ando, Esfandiar Bandari, Andrew Csinger, Heidi Dangelmaier, Elliot Drobner, Mike Horsch, Maxime Kalfon, Jian & Ying Li, Steve & Sarah Mason, Valerie McRae, Manny Noik, Sue Rathie, Greg Rowland, David Sidikover and Diana Weaver.

There were many experts who were happy to help when approached. To name a few: Donald Acton, Barry Brachman, Greg Burns, Ian Cavers, Frank Flynn, Nadine Hofmann, Ming Lau, Marc Majka, Rick Morrison, George Phillips, Peter Phillips, Rick Sample, Bernd Stramm and Deborah Wilson.

And how does one adequately thank the secretarial staff? They know all the answers and they know all the funny lines. Here they are. Gale Arndt, Evelyn Fong, Theresa Fong, Sunnie Khuman, Carol Whitehead, and Grace Wolkosky.

Finally, in addition to my thanks, I offer my apologies to those whom I have forgotten to mention.
Chapter 1

Introduction

Parallel processing has been used to advantage for countless centuries. As discussed in [2], current supercomputers are incapable of analyzing static images and doing real-time analysis of motion, “Yet biological early vision processes are clearly able to perform the computations by exploiting the inherent parallelism of visual inputs in a truly concurrent fashion.”

The internal speeds at which a biological system functions are slow compared to a typical CPU. According to model parameters used in [1], neuronal signals travel typically between 1 and 10 m/sec, which is seven orders of magnitude slower than the speed of electric signals used in computers, and the refractory period (delay between synapse firings) is modelled at 10 ms, which is five orders of magnitude slower than the clock speeds of modern CPU’s.

Obviously, the combining of the best of both worlds – massive parallelism and the internal speed available through current technology – offers computational power well beyond our present capabilities.

A parallel program runs on a network of processors. In addition to the hardware, system software is required to help the programmer develop the program. This situation
Chapter 1. Introduction

is analogous to the use of high level languages to program a standard computer, rather
than machine instructions or assembly language. The added complications centre around
deciding which processors should run which pieces of code, and along which channels they
should communicate. This is known as the mapping problem, and it is NP-complete; some
of the difficulties involved in mapping are discussed in Berman [11].

1.1 Description

This thesis describes the TMAP system (Transputer MAPping tool), a parallel program-
ing environment, specifically designed and tailored for the multiprocessor network at
the Computer Science Department of the University of British Columbia; the system is
designed, however, so that porting to another site would require a minimum of time and
effort.

The hardware that comprises our network is centred around 74 transputers (cf [14]).
Each transputer is a CPU with its own memory, and 4 serial data links. Memory is
not shared amongst the transputers; inter-node communication is effected solely through
the data links, which may be wired directly to each other. However, most data links
are wired to crossbar switches. Our network contains 10 of these switches. Each switch
has 32 data links and is capable of creating, under software control, 16 bidirectional
connections. Other transputer links are used for I/O with the external world. Due to its
size and to various hardware constraints, our network is irregular and complex. A more
complete description is given in §2.1.

One property of TMAP is to hide this complexity from the user, and present the user
with a manageable interface which is logically consistent with the machine architecture
and topology. Moreover, the TMAP system incorporates the fact that our network is made up of a variable number transputers and software switches, and provides software tools to configure and control the switches and manage user level access; cf §2.2.

The scope of the TMAP project ranges from user program design to allocation of hardware resources and the mapping of the processes onto these resources in a way that is efficient with respect to the interconnection potential of the hardware.

The building of TMAP required the integration of two major pieces of software. The first is Prep-p [12], which we modified and used to parse the application program’s process graph, and to partially map the processes into the network. The second is Trollius [13], which is the operating system that we run on our processors. We rely on Trollius to boot the machine, load programs and do high-level message passing. These aspects are elaborated on in §4.3.

The lowest level software model supported by our hardware is that of a collection of processes who send messages to one another; indeed, the transputer has hardware support for context switching and inter-process communication. The degree of synchronously of the communication is controllable from the application program on a per message basis via buffering. So, this is not communicating sequential processes in the strict sense of Hoare [21], where there is a static collection of processes with necessarily synchronous end-to-end communication.

The software model, of communicating processes, is used by TMAP to present the programmer with a virtual machine, thus relieving the programmer from needing to know about the specific architectural details of our configuration. An application program is described to TMAP as a graph, where each node is a process, and the labeled edges are
the channels of communication that the programmer may reference in the application source code, for inter-process communication.

A guiding light for the development of TMAP was to obtain coordinate-free communication without loss of efficiency in the run-time code. Frequently, after mapping a process graph to processors, channels end up being of three types.

- **Local** – The 2 processes are on the same processor.

- **Adjacent** – The 2 processes are on processors having a direct physical communication link.

- **Distant** – Communication between the 2 processes involves hopping through intermediate nodes.

Distant communication is expensive using Trollius, orders of magnitude greater than with channels of the first two types. Yet, even in these two cases, it is difficult to take advantage, in Trollius, of the efficient low level communication. The network information needed to do this is not immediately known to the programmer, but is known only after the mapping is completed. Given the immense gain possible for using low level communication, it was important to develop a system that would, almost completely, take care of the bookkeeping to allow the programmer to easily make use of the efficient communication offered by the hardware. This has been accomplished by TMAP.

The virtual machine, as seen by the application programmer, is a black box on which a set of communicating processes may execute. These processes constitute the parallel program. It is important to note that this software model is at a sufficiently low level that TMAP is able to take full advantage of the most efficient communication supported
by the hardware. Moreover, this is accomplished through high level send/receive library routines, which are easy to use.

This approach to programming allows applications to be

- **Scalable** – A single parameter could increase the size of the process graph,
- **Coordinate free** – Inter-process communication uses only the channel names occurring in the process graph,
- **Topology independent** – An application need not be recompiled, even if the network hardware interconnections are changed, and
- **Portable** – Once TMAP is ported to another platform, previously written applications for TMAP are able to run at the new site.

To keep the scope manageable, TMAP does not deal with the routing of distant channels, other than by using the Trollius router. With the advent of the T9000\(^1\) transputer [3], even distant routing will be hardware supported, although placement will still be important.

Despite this aversion towards distant communication, broadcasting, which is not implemented in this version of TMAP, may be naturally added to the system; cf §9.2.1. A broadcast occurs when one process sends the same message to a group of processes, often dispersed throughout the network.

The TMAP system design is detailed in Chapter 4. A complete TMAP session allows a user to compile, link and load the code which implements a parallel program, and

\(^1\)Formerly known as the H1.
consists of the sequential execution of 7 modules. As input to the first module, the user describes the logical structure of the application program. The successive modules allocate resources, set up structures to implement the requested channels of communication, and eventually load the user's code onto the processors.

Another important property of TMAP is that the programmer is able to specify as much hardware/network information as desired; the system would supply intelligent choices for whatever the programmer omits. This "freedom of information" does not encumber the programmer, and offers maximal flexibility in using the environment.

This property was implemented using a specific design methodology. Each module expects certain text files as input. All of these, except for the Graph Description File (cf §4.4) and the user's source code, are generated by the system, and contain only the minimal information needed by subsequent modules. The user may modify these output files as necessary, before they are used by the system. As an aid in bookkeeping, the system supplies a shell script with which to invoke TMAP with the specified file changes and flags; the user is expected to edit this script.

1.2 Other Work

Several topology-independent parallel programming environments for transputers have heretofore been developed. However, none of these systems may be easily ported to our multicomputer, and none of them automatically configure switchable network connections. Moreover, with TMAP, there is no restriction on the language being used to code the application.
Four of these systems are described, briefly, below. We have not had direct experience with these systems; our summaries are based on descriptions that have appeared in the literature.

The popular system Express [5] is not elaborated on here, as it is not topology-independent – the user routes messages by specifying the physical node numbers.

**Helios**

Helios [9] is actually an interactive distributed operating system, implementing a UNIX type interface that may run on the nodes. It allocates resources for a user program according to what is currently available in the network; however, the network is assumed to be static, i.e. the transputer interconnections are not switchable. The situation is reversed in TMAP – resource allocation is done statically (cf §4.4), but the transputer interconnections are set to best accommodate the user’s requested communication configuration using the programmable switches (cf §4.2). Helios uses its own C compiler/linker.

**A Neural Network Modeller**

The system [6] is designed to model modular neural networks on a transputer network. It is based on CSTools ([8]), where standard C or Fortran may be used by the programmer to implement parallel processes. The Compiler step translates the user’s description of the neural network, to be used as input for the next module, the Splitter, which decomposes the problem. The third and last part is the Simulator, which loads the user’s code, as directed by the Splitter. The system is claimed to be quite flexible, and should be able
to handle problems which are not in the neural network domain. I could not ascertain whether the system deals with switchable transputer interconnections.

MARC

The system [4] accepts only application programs written in occam [10]. This language is a somewhat higher level replacement for the difficult RISC assembler of the transputers. It is limited as a production language, as it has a small selection of data types, no pointers, no stack, and no support for recursion or dynamic process creation; it does have built-in message passing and synchronization routines, which actually guided the design of the transputer, which, therefore, is able to efficiently implement the occam constructs.

By using occam, the user's code is automatically decomposed into a fine\(^2\) set of communicating processes. However, the placement of these processes into the network is left to the user (or to the environment in which the user is working). Moreover, I believe that at this stage of our understanding of the mapping problem, such a fine-grained description of the user's program is not sufficiently useful to offset the work required to place them into the network.

MARC uses the transputers themselves, in a distributed fashion, to come up with a mapping for the processes, using a method based on diffusion. I believe this to be a good idea. Our network at UBC has variable connections; it would be interesting to devise such a method on our hardware. In TMAP, all the work, except for actually running the user's code, is done by our host, a SUN SPARC station.

In order to avoid deadlock, MARC considers only those process graphs that admit

\(^2\)An individual instruction may actually be a process!
an Eulerian path. Messages are routed, bidirectionally, along the path. A short cut may occur if a node would be visited twice. In TMAP, the routing of channels which are between processes that are neither on the same nor neighbouring nodes is handled by Trollius. Although this is not guaranteed to be deadlock-free, this aspect has not been a problem for us.

As in Helios, the system does not automatically configure the network to best accommodate the user program – the wiring must be done beforehand.

Shea

The system described in Shea [7] configures occam programs to run on a transputer network. The user is expected to contract (or partition) the processes to number no more than the number of available processors, and the system then places the processes into the network using a greedy algorithm. The system is centred about a routing mechanism which eliminates deadlock and maintains synchronously of all channels, even those that are dilated.
Chapter 2

Hardware and Maintenance Software

Our multicomputer system consists of various hardware components. These are managed by low level interface routines, by which all other software may access and control the various components. Section 2.1 describes the hardware in some detail. This sets the stage for describing the maintenance software in §2.2.

Our numbering system is quite simple. If there are $N$ pieces of a particular hardware type, they are numbered $0 \ldots N - 1$.

2.1 Hardware

The heart of our network consists of 74 INMOS transputers (cf [14]). Each is a 32 bit CPU (model T800) with either 1 or 2 Meg of RAM, and 4 data links for internode communication. There are also 10 C004 programmable $32 \times 32$ crossbar switches, each having 32 data links and 1 control link (cf [14]). Pairs of data links on a C004 may be connected by sending appropriate software commands to the control link. Most of the transputer links in our system are connected to data links of the crossbars. In this way, transputer link interconnections may be varied through software commands, as requested by a user.
### Table 2.1: Ports to access the network

<table>
<thead>
<tr>
<th>Port</th>
<th>Function</th>
<th>Location</th>
<th>Connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Data</td>
<td>CSA board 0, link 0</td>
<td>Transputer 0, link 1</td>
</tr>
<tr>
<td>1</td>
<td>Data</td>
<td>CSA board 0, link 1</td>
<td>Transputer 2, link 1</td>
</tr>
<tr>
<td>2</td>
<td>Data</td>
<td>CSA board 0, link 2</td>
<td>Transputer 4, link 1</td>
</tr>
<tr>
<td>3</td>
<td>Data</td>
<td>CSA board 0, link 3</td>
<td>Transputer 6, link 1</td>
</tr>
<tr>
<td>4</td>
<td>Not used</td>
<td>CSA board 0, link 4</td>
<td>—</td>
</tr>
<tr>
<td>5</td>
<td>Not used</td>
<td>CSA board 0, link 5</td>
<td>—</td>
</tr>
<tr>
<td>6</td>
<td>Data</td>
<td>B011 board</td>
<td>Transputer 73, link 0</td>
</tr>
<tr>
<td>7</td>
<td>System</td>
<td>CSA board 1, link 0</td>
<td>Crossbar 0, link 30</td>
</tr>
<tr>
<td>8</td>
<td>System</td>
<td>CSA board 1, link 1</td>
<td>Crossbar 0, control link</td>
</tr>
<tr>
<td>9</td>
<td>Data</td>
<td>CSA board 1, link 2</td>
<td>Crossbar 1, link 18</td>
</tr>
<tr>
<td>10</td>
<td>Data</td>
<td>CSA board 1, link 3</td>
<td>Crossbar 1, link 19</td>
</tr>
<tr>
<td>11</td>
<td>Not used</td>
<td>CSA board 1, link 4</td>
<td>—</td>
</tr>
<tr>
<td>12</td>
<td>Not used</td>
<td>CSA board 1, link 5</td>
<td>—</td>
</tr>
</tbody>
</table>

### Table 2.2: Reset Components

<table>
<thead>
<tr>
<th>Component</th>
<th>Port</th>
<th>Size</th>
<th>Transputers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>18</td>
<td>0 1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>18</td>
<td>2 3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>18</td>
<td>4 5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>19</td>
<td>6 7 8</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>1</td>
<td>73, on the B011 board.</td>
</tr>
</tbody>
</table>
The host machine through which the network is accessed is a SUN SPARC workstation, having 13 ports. Each functioning port is connected to a data link of either a transputer or a crossbar. Two ports, 7 and 8, are used exclusively for setting the crossbar switches. We have experienced problems with 4 of the ports, so we do not use them. The remaining ports may be used by application programs to implement communication between the transputers and our host. Table 2.1 contains the details. There are two types of interface boards – the B011 board of INMOS ([15]), and 2 boards from Computer System Architects ([16]).

Aside from the host, there are 2 other physical locations which house our network. The old box contains 9 transputers and 2 crossbar switches, and the new box contains 64 transputers and 8 crossbars.

The network is divided into 5 reset components, as summarized in Table 2.2. Each component has a root transputer that is wired directly to a port. The remaining transputers in a component are arranged in a linear reset chain starting with the root; when the port is reset, all the transputers in the corresponding component are reset, and are ready to be loaded for the next session. In this way, our system supports up to 5 concurrent users, who have complete control over the transputers allocated to them. A user may be allocated as many components as desired.
2.2 System Software

2.2.1 Resources

Our goal for system management is to allow simultaneous access of the multicomputer by several users. We do not run a multiuser operating system on each transputer, so we decided that each user would “own” a certain subset of the transputers, and the system would guarantee that users be protected from one another.

Each piece of network hardware has a number of hard links which may be connected to other links. To implement our management policy, we noticed that it suffices to view each link as the unit of resource allocation. In a natural way, the link allocation strategy varies according to the piece of hardware being discussed, as explained in the following paragraphs.

On a transputer, a user gets either all the links, or none of them; this is implemented by allocating the transputer – the links are allocated implicitly. Actually, transputer allocation is even coarser; the unit is an entire reset component. As an example of the irregularity in our network, there is 1 transputer, number 8, for which link 3 is unavailable. It is connected to a specialized circuit that implements a real-time monitor, to be used with the TMON system [19].

On our host, only data ports are allocated, and this is done individually. Even if one user gets all the ports, other users may still access the host, but they will be unable to access the transputers.

On a crossbar, 3 data links (and the control link) are reserved for system use, for
the setting of the switches. This is explained in detail in Chapter 3. The remaining 29 data links may be allocated to users. Some data links on some crossbars are wired to data links on other crossbars. This was done to enrich the set of possible interconnection networks. These inter-crossbar wires are treated as a unit for allocation purposes – a user is allocated either both links or neither link.

2.2.2 Hardware Database

As can be seen from the hardware description, the system and interconnections are complex and irregular. All of the link-to-link connections are maintained, hierarchically, in a text hardware configuration file. The information is converted by lex and yacc into a binary form, which is made available to system maintenance routines.

Note that in many systems, the software often depends on some regular link to link connections. For example, link 0 of each transputer is connected to link 1 of the next, in order to form a chain. This regular use of links is not present in our system, nor should it be.

2.2.3 Ports

Simultaneous access of a particular port by more than 1 process is prevented through the use of lockfiles, there being 1 for each port. The standard functions of reading and writing with timeout, and resetting are provided.
2.2.4 Crossbars

The 10 crossbars described in §2.1 are managed, at the lowest level by the software described in Chapter 3. In addition, each of links 0...28 may be locked by a user (The other links are reserved for system use). The status of these locks, for all the crossbars, is maintained in a single lockfile.

2.2.5 Session Setup

The first step in setting up a prospective user's session, is locking the required resources. If any such resource is unavailable i.e. locked by some other user, all previous locks, if any, obtained by the prospective user are released, and the user must restart the session.

This simple-minded strategy for resource management does make deadlock very unlikely. The possibility for deadlock still exists, since there is no a priori order in which to request the ports; however for deadlock to occur, the sequence of restarts for each user would have to be carefully timed to lock a resource the other has not yet obtained.

Not surprisingly, deadlock has never occurred. Moreover, even the remote possibility of deadlock may be eliminated, as was done for the crossbar links, by using a second lockfile for the ports.
Chapter 3

Crossbar Control

This chapter describes our current method of setting the crossbar switches. The method was chosen for 2 reasons. Each crossbar is treated the same way, so there is simplicity in the symmetry. More importantly, it minimized the number of wires necessary to cross between the old and new boxes of our network (cf §2.1); at that time, this seemed like an important criterion.

The major disadvantage to this method is that 3 data links on each crossbar are unavailable to users for the forming of networks. Another theoretical problem is that the algorithm is exponential in time with respect to the number of switches being used; in practice, the numbers are small, and the time appears to be instantaneous.

The method is to arrange the switches into a linear control chain, as described below. The algorithm is optimal, based on a complexity analysis, and was implemented using the crossbar switches described in §2.1.

In a larger system, this method could be used in conjunction with the method of “fanning out” from a single switch to the control links of other switches (at most 31 in a single fan). Fanning out is done in constant time, requires a dedicated switch and probably several long wires.
Chapter 3. Crossbar Control

The discussion that follows relates to our system by taking $N$ to be 9, port A as 8, and port B as 9. The specific connections are described in Table 2.1.

3.1 Introduction

The $N+1$ switches are labeled $C_0 \ldots C_N$.

Each has 32 data links, labeled 0 \ldots 31, and 1 control link, denoted as 'L'. Any 2 data links may be connected to each other, by sending the appropriate command to the crossbar through the control link. In this way, up to 16 bidirectional connections may be formed. Link number $i$ on switch $C_n$ is denoted $C_{n,i}$.

Three data links, say, 0, 1 and 2, are chosen, and together with the control link, are used solely to configure the switches in a chained fashion. For each switch, except the last one, link 2 is wired to the control link of the next switch. Port A is wired to $C_0.L$, and port B is wired to $C_0.1$. For each $i$ from 0 \ldots (N-2), $C_i.0$ is wired to $C_{i+1}.1$. This configuration is illustrated in Figure 3.1.

A switch is said to be controlled if its control link is communicating with either port A or port B, while using only links 0, 1 and 2 of any of the switches to attain the communication path.

Due to the connection of port A, switch $C_0$ is always under immediate control of the host. For $n > 0$, switch $C_n$ is controlled when port B is connected, via crossbar connections, to $C_n.L$. There is only 1 way for this to be the case; the path to control $C_3$ is shown with double arrows in Figure 3.1.
Chapter 3. Crossbar Control

The remaining links of each switch are used to configure the multiprocessor system; one must get control of each switch, and issue it whatever extra commands are necessary to effect the network configuration.

In §3.2, an algorithm is described for doing this. In §3.3, this algorithm is shown to be optimal, under the constraints of how the switches are hardwired together.

3.2 Description of the algorithm

The main algorithm is described in terms of Procedures 3.1 and 3.2, which call each other recursively. It will be seen in §3.3 that Procedure 3.3 requires $2^N - 1$ switch connections, and that this is optimal; of course, this number does not include the extra network configuration commands that are issued to each switch. The main algorithm could be implemented with a simple recursive function, but then it would be awkward to issue each switch its network configuration commands.
Procedure 3.1 (Get_Control_Absolute) *No assumption is made about the current switch settings.*

Input: n — The switch number to get control of.
Output: port — The port number to use to send commands to the switch.

begin
    For i = 0 to n do Get_Control_Next(i, port)
end

Procedure 3.2 (Get_Control_Next) *The previous switch is assumed to be controlled, except, of course, if \( n = 0 \).*

Input: n — The switch number to get control of.
Output: port — The port number to use to send commands to the switch.

begin
    If \( n = 0 \) , set port = A and return.
    Set port = B.
    Connect links 1 and 2 in switch \( n - 1 \).
    For i = n - 2 down to 0 do
        begin
            Get_Control_Absolute(i, port)
            Connect links 0 and 1 in switch i.
        end
    end
end

Procedure 3.3 (Configure_Switches) *Each switch is controlled, in turn, and issued its configuration commands.*

Input: SwitchSettings — A file containing the switch settings to configure the network.

begin
    For i = 0 to N do
        begin
            Get_Control_Next(i, port)
            Use the data in SwitchSettings to issue the network configuration commands to this switch. Specifically, switch number 'i' is issued its commands through port number 'port'.
        end
    end
end

Figure 3.2: The 3 switch setting procedures.
3.3 Derivation of the algorithm

Let \( u_n \) be the number of switch connections that are needed to guarantee getting control of switch \( C_n \); this is the complexity of Procedure 3.1. As already noted, \( u_0 = 0 \). For \( n > 0 \), the previous switch must be controlled and issued 1 command to connect links 1 and 2, as depicted in Figure 3.1. Thereafter, all the previous switches must be controlled, in decreasing order, and issued 1 command to connect links 0 and 1. Thus, the following recurrence relation holds.

\[
    u_n = u_{n-1} + 1 + u_{n-2} + 1 + \cdots + u_0 + 1 \tag{3.1}
\]

From this follows

\[
    u_{n+1} = u_n + 1 + u_n = 2u_n + 1 .
\]

The solution to this equation is, for all \( n \geq 0 \),

\[
    u_n = 2^n - 1 . \tag{3.2}
\]

Equation 3.2 expresses the values that an algorithm would like to achieve. Such an algorithm is described with the help of an intermediate sequence. Let \( v_n \) be the number of switch connections necessary to get control of \( C_n \), given that \( C_{n-1} \) is controlled; this is the complexity of Procedure 3.2. Of course, \( v_0 = 0 \). For \( n > 0 \), the relation is

\[
    v_n = 1 + \sum_{i=0}^{n-2} (u_i + 1) \\
    = 1 + \sum_{i=0}^{n-2} 2^i \\
    = 2^{n-1} . \tag{3.3}
\]
A consequence of Equation 3.3 is that

\[ \sum_{i=0}^{N} v_i = 2^N - 1 = u_N. \]

This last equation shows that Procedure 3.3 has the claimed complexity, and that this is the best lower bound, \( u_N \).
Chapter 4

System Design

The user program is described to TMAP as a logical collection of communicating processes. The user supplies this decomposition directly, or it would be obtained from TRES [18], another project at UBC, under development. This information is what we felt to be the minimal necessary to implement a configured system to efficiently run the user's program.

Various graph structures are used throughout TMAP, and are described as they arise in the following discussion.

4.1 The Front end

As mentioned in §1.1, we have integrated Prep-p [12] and Trollius [13] into our system.

The first module in Prep-p parses the user's description of the application program. The description is in terms of a process graph. The labeled edges correspond to channels of communication. The second module assigns communication weights to the channels mentioned in the process graph. The third module contracts (or partitions) the process graph to have a sufficiently small number of nodes i.e., no more than the number of
available processors. In this way, Prep-p separates the mapping problem (cf Chapter 1) into 2 stages – contraction followed by an embedding.

The basic system design of TMAP is modeled after Prep-p, e.g. using text files to communicate from module to module. In fact, the first 3 Prep-p modules have been modified to work in TMAP; cf §4.4 and Appendix B. TMAP diverges from Prep-p since the target hardware architecture of Prep-p is the CHiP [22], a synchronous multiprocessor, whose I/O methodology is quite different from our system's.

4.2 User Input

The main user input is a weighted undirected process graph. Each node of this graph is called a virtual node (Vnode), in keeping with the Prep-p nomenclature, and represents a process that will be executing on some processor in the network. In addition, each edge has 2 labels, so that each vnode may have its own name for the edges incident to it. In this way, a vnode now has a name of its own choosing for each of its neighbours, and this is used to implement the topology-free communication described in Chapter 6 – each edge of the graph represents a communication channel.

Each vnode is implemented as an executable file. The user specifies whether it is to be run on the host or on a transputer. Thus, there are 2 types of vnodes, and are described using Trollius [13] terminology – ITB “In the box”, and OTB “Outside the box”. This virtual graph (Vgraph) is also weighted, each edge being labeled by an integer estimating the amount of communication that the user expects to occur along that edge.

We decided that each edge should be bidirectional, as a transputer link may carry
information in either direction, and this tends to simplify the graph.

As mentioned in §1.1, Prep-p is used as a basis for the front end of TMAP. Specifically, we use the first 3 modules of Prep-p to contract a process graph to a specified number of nodes. Changes that we made to Prep-p include specifying the processor type on which a process is to be run, contracting only the set of processes targeted for transputers, allowing the specification of more than 1 communication channel between 2 processes, and redefining how I/O is to be done. These changes were necessary, as the target hardware architecture of Prep-p is so different from our own.

Further information that may be supplied by the user includes the following.

- **Communication channels** (edges in the process graph). The user may use these channel names in the application source code for inter-process communication. The semantics are described in Appendix A.2. These names are assigned values when the executables are loaded onto the network. This relieves the user from needing to know about the hardware topology, the numbering of the links, and eliminates the need for recompilation when the topology changes. Also, only 1 copy of the executable need be maintained on disk, rather than all the patched copies that get loaded onto the various nodes!

- **Number of transputers to use.** The contraction module partitions the ITB vnodes into disjoint groups, each being called a real node (Rnode). Each transputer will receive at most 1 rnode i.e. each transputer will receive the executables associated with at most 1 rnode. The edge weights are used to guide the contraction. The TRES project deals with the question of determining an optimal number of nodes, as well as other parameters such as topology, on which to run a specified set of user
• **Number of ports to use.** In our system, it is the number of ports that limits the number of concurrent users of our system. For this reason, the default number of ports is one.

• **Channel usage (edge weights).** These weights are estimates by the user as to how much the communication channels will be used.

It is recommended that the user specify the first 3 of the above 4 categories, especially the first two. The user may also specify how to connect the transputers via the software configurable crossbar switches, how to contract the vnodes to rnodes, and how to place the rnodes onto the network. Some of this is accomplished with the help of intermediate languages. Obviously, most users would be happy to leave this work to the system.

### 4.3 The Back end

As mentioned in §1.1, a user's program, loaded by TMAP, is designed to run under the Trollius operating system. Trollius follows, at a low level, the same design constraint as TMAP viz. hiding information while allowing the user to specify as much detail as desired.

Trollius also employs user-supplied/modifiable text files, e.g. the network configuration file (bnail), and this fits well into the TMAP design. However, it is a major chore to write up a bnail file in our system, as the interconnections are, typically, quite irregular. Furthermore, any changes to our system e.g. a minor rewiring, would require any affected bnail files to be edited and changed. The TMAP system automatically creates any bnail
files that are needed.

Trollius implements network-wide routing for message passing. Under Trollius, to send a message to another process, the Trollius node number of the destination process must be known to the sending process. When the destination process is on an adjacent node, or on the same node as the sending process, there are more efficient message passing routines to use. However, to make use of them, the programmer must be acutely aware of the interconnection topology being used on the transputers, and the mapping of processes onto this topology.

With Trollius, if the user chooses the same node numbers to use in various topologies, then the same code may be run without recompilation; however, the routing of messages will probably be inefficient, involving hops across nodes for messages to reach their destinations. On the other hand, if the user processes are loaded onto the nodes in a way which is logically more consistent with the communication patterns of the program, then the code must be recompiled with the correct network information; knowing this information is a burden for the user. The elimination of this problem was a chief motivator in the design of TMAP.

The TMAP project may be viewed as extending the information hiding of Trollius to include not needing to know the Trollius numbers of nodes running specific processes. However, the design of our system is such, that it is amenable to modification for use with other low level transputer management systems, such as Logical Systems' ld-net [24]. This is explained in Chapter 9.

Only two functions were added to the Trollius libraries in order to implement the send/receive capabilities using the channels defined in the graph of vnodes. These are
described in Chapter 6.

4.4 Outline

TMAP consists of a sequence of modules which, normally, are executed in order.

1. Vgraph module. User input consists of a graph description file (GDF), which is processed largely by the GraphCom module of the Prep-p system. This GDF specifies the user processes, along with their communication channels, using a convenient graph description language. The source code for a process may reference a structure variable having the same name as the specified channel. At load time, the fields of this structure variable are patched to contain the Trollius number of the destination node and an integer event specifying the particular process on the destination node.

2. Weight module. The user may use the XX language of poker [23] to describe, for each process, those aspects of the process that the user deems appropriate. This skeleton code is scanned by the CallXX module of Prep-p to generate weights for the edges.

3. Resource module. The resources in question are which transputers and ports to use. The user may have specified nothing about this, or partial information, or completely the desired hardware. Partial information may be simply how many nodes and ports to use, or even specifying some of them. In any case, we use the method of binary integer linear programming to satisfy the outstanding resource requirements; cf §7.1. Our current network is divided into 5 reset components; cf
§2.1. Each component is assigned a binary integer variable to represent availability of the component and port. The objective function’s main goal is to minimize the number of components being used, as it is the number of components which limit the number of concurrent users for our system.

4. *Contract module.* The contraction module of Prep-p is used to contract the vnodes to the required number of rnodes, as explained in §4.2.

5. *Place module.* The embedding of rnodes into the allocated transputers, and the formation of transputer connections, are accomplished by adapting a greedy algorithm of Smedley[20].

6. *Setup module.* Up to this point, the work done by TMAP has been independent of the low level software being used, in this case Trollius. The purpose of the Setup module is to minimize the amount of work that the next module, Load, must do. To this end, the Setup module organizes the information that is needed by the Load module. This includes the creation of the Trollius files *bnail.* (network topology) and *route.* (routing).

7. *Load module.* The resources are locked exclusively for the user. If any of the resources is not available, all locks for this user are released, thus avoiding deadlock, and TMAP is exited; otherwise, the executable files are loaded onto the network.
Chapter 5

Sample Session

The example in this chapter is intended to help the reader form a single picture of the TMAP system, and does not cover all the features offered by TMAP, such as command line parameters that each vnode may receive at load time. Complete documentation may be found in the appendices.

The ITB nodes form a ring. Each experiment consists of passing a message of a specified length around the ring a specified number of times, using one of 3 different types of message passing, and printing how long it took to do this. The 3 levels of communication are described in Appendix A.2.

Before each experiment, a control packet, specifying the 3 parameters, is sent to each node. Appendix D has all the code for this example.

Here is the GDF that is used:

/* This is a Graph Description File. It is based on the GraphCom module of Prep-p. The first processing done on this file is by the C preprocessor, cpp, whose output is directed into GraphCom. */

/* Even though Prep-p is able to handle special graph types more optimally, this ring is specified as 'irregular', for simplicity. */
irregular
{
    /* The user may define variables and initialize them. */
    /* This first variable is relevant to this example, only. */
    /* It is the length of the ITB ring. */
    /* To keep things uncomplicated in this example, */
    /* it must have the value at least 3. */
    length = 10;
    /* These 2 variables must always be defined for Prep-p. */
    nodemin = smallest number of a vnode.
    nodecount = total count of vnodes.
    */
    nodemin = 0;
    nodecount = length + 1; /* 1 more for the host node. */
    /* These 3 variables are specific to TMAP, and */
    /* take on default values if they are not defined: */
    OTBmin = -1, i.e all vnodes are ITB.
    TRANScount = 18. /* 18 is a good number for the UBC network. */
    PORTcount = 1.
    */
    /* All vnodes numbered above or equal to OTBmin will be loaded OTB. */
    OTBmin = length;
    /* At least this number of transputers will be allocated by the */
    /* Resource module. */
    */
    TRANScount = length;
}

procedure Host
    nodetype : { i == OTBmin }
    port root : {0}

procedure root
    nodetype : { i==0 }
    port host : {OTBmin}
    port next : {1}
    port previous : {length-1}
The preceeding GDF specifies that the ITB ring has 10 nodes, with each process being connected to its neighbours by channels it knows as "next" and "previous". Each node procedure is a template for 9 vnodes numbered 1 ... 9. The root procedure represents only vnode 0, which has an additional channel called "host" with the sole OTB process, number 10, represented by the procedure Host. Figure 5.3 depicts this arrangement.

The rest of this chapter explains the steps to actually run the program. Ideally, the command \texttt{tmap} is all that would be necessary. Unfortunately, reality sets in.

- We are insisting on a perfect embedding of the communication graph specified in the GDF; the Contract module must be told not to do any contraction.
- The embedding of the "contracted" graph into the hardware is based on heuristics; cf §4.4. In this example, it turns out, the algorithm must be told the mapping of
a few of the nodes, in order to avoid dilations. A dilated channel does not support the 2 lower levels of communication. Even so, most nodes are taken care of by the system, and at no point need one deal with link numbers. This is the only point where the user would need some familiarity with the hardware.

- The Load module must be told that the host process is to be run in the foreground, as it is accepting input from the keyboard. By default, host processes are run in the background.

- For reasons of efficiency, the Load module does not check that an executable to be loaded onto the network is up to date with respect to the user's source code. The user must ensure this and prepare the makefiles before invoking the Load module, in the manner described in Appendix A, paragraph "USER CODE". However, if an executable is not present, the Load module will automatically make it.

The complete command line is

```
tmap  -C n  MapInit=MapInit.1  LoadLst=LoadLst.f
```

The syntax is described in Appendix A.
As explained in §4.2, a program consists of a set of sequential processes, which are running on the various processors. During its execution, each process will be sending information to, and receiving information from, other processes in the network. The two communication routines are \texttt{ch\_send} and \texttt{ch\_recv}. Their semantics, basic functionality, and 3 levels of communication are described in Appendix A.2. The remainder of this chapter is concerned with the implementation of these 2 routines.

The channel structure consists of 2 integer fields viz. \texttt{node} and \texttt{id}. The \texttt{id} field encodes whether the channel is local i.e. whether the 2 ends of the channel are on the same physical node, and this is known only after the contraction of the graph of virtual nodes. The first time a channel is used, it may initialize itself; however, the concerns are different for local and non-local channels.

### 6.1 Events, Virtual circuits and ID's

Trollius uses an integer called an \textit{event} to distinguish receivers on the same physical node; thus, 2 integers are used for the routing of messages – the node number of the destination, and an event.
Trollius does offer an optimized level of communication, called a virtual circuit. The increased efficiency results from fewer inter-process calls being made. Normally, on each node, the link proprietors, the router, the kernel, the buffers and the user must coordinate. When two communicating processes set up a virtual circuit, no other events are allowed to use the hardware links that make up the circuit.

The lower 27 bits of the id field of the TMAP channel structure contains an integer which is called the channel id. These numbers are generated as the GDF is parsed; cf §4.4. The Trollius event which the channel uses for network level sending is 0x7FFFFFFF − id. In this way, TMAP uses only high positive events, allowing the user to use the remaining positive events (negative events are reserved for the Trollius kernel).

Unfortunately, this event cannot be safely used for receiving from a virtual circuit, since Trollius requires that the lower 4 bits of such events, being used by a process, be unique; these 4 bits are used to index into a virtual circuit descriptor table which has 16 slots. Fortunately, the largest degree of any of our nodes is no more than 16 – for a transputer it is 4, and for our host it is currently 7 (the number of ports available to a user). Each link may support only 1 virtual circuit at a time.

When the TMAP communication routines are setting up a virtual circuit, the table is searched for a high-numbered available slot, say number i. The event which is used for receiving on the virtual circuit would then be 0x7FFFFFFFi, corresponding to a channel id of 0xF − i. For this reason, the assignment of channel id's begins at a number greater than 0, currently 10.
6.2 Local Channels

The purpose of initializing a local channel is to synchronize the two ends for communication that would bypass Trollius services. Thereafter, the user would be able to take advantage of this option by setting the flag parameter in the communication routines.

For local channels, only 2 levels of communication are implemented. The Trollius virtual circuit is not used; if requested TMAP uses the hardware level, instead. The main reason for this design decision was to not take up valuable slots in the Trollius virtual circuit descriptor table. This table must be used to effect low-level communication for the non-local channels, as described in the next section.

Initialization is relevant to ITB nodes, only. A local OTB channel never initializes itself. This decision was taken for this first release of TMAP, since local OTB channels were not deemed sufficiently important. A future version of TMAP may use sockets or shared memory to implement "hardware" communication for local OTB channels.

The following initialization procedure is invisible to the user, and is handled entirely by the TMAP communication routines.

When an ITB local channel is first used, the sender tags on to the message the address of the node field, and this address is placed into the node field at the receiver's end. The sender's node field may now be used for synchronization using the in/out machine instructions of the transputer. The sender's id field is coded to indicate that the node field is the channel synchronization word, while the receiver's id field is coded to indicate that the node field contains the address of the channel synchronization word.
6.3 Non-local Channels

Typically, to route a message at the network level, Trollius sends a message to the router daemon, which responds with the hardware link to use. Each process streamlines this protocol by caching a table of links for intended destinations.

The purpose of initializing a non-local channel is to store, in the id field, the link number which is being used to route the channel with the other end. Once this is done, the communication routines may use the datalink layer of Trollius, rather than the slower network layer.

On a transputer, the link number is between 0 and 3; on the host, the link number is between 0 and 10, these being the port numbers (cf Chapter 2.1). Four bits of the id field are reserved for this.

If the channel is not dilated i.e. the 2 ends are on adjacent nodes (this is known after the contracted graph has been placed into the hardware), the link number is patched into the channel structure as the executable is being loaded onto the network. This is accomplished by the TMAP front end to the Trollius loader. For a dilated channel, initialization is done at run-time, by making a call to the Trollius router. This is done automatically by the TMAP communication routines.

TMAP does not support hardware communication over a dilated channel, and Trollius recommends that a virtual circuit not be used over a dilated channel, because of the likelihood of deadlock. So, the following discussion, on the 2 lower levels of communication, refers to channels between nearest neighbours.
The first step in setting up a hardware communication is the obtaining of a virtual circuit. The user initiates this by setting the appropriate bit of the flags parameter to the TMAP communication routines. The consequent flurry of activity, outlined below, to implement this request is invisible to the user.

The sender obtains an event for receiving along the virtual circuit as explained in §6.1, and tags it along with the message. The receiver will use this event for sending along the virtual circuit. It is at this time that the receiver sends the sender an event that the sender should use for sending along the virtual circuit!

This convoluted logic occurs only during the setting up and dismantling of a virtual circuit. Once established, the user may choose to efficiently use the virtual circuit or the hardware, by setting the appropriate bit in the flag parameter of the communication routines, assuming that both ends are ITB; if one end is OTB, hardware communication is not an option.

Chapter 9 contains the results of experiments comparing the throughputs of the 3 levels of communication.

6.4 Channel Properties

A channel may be shared amongst different processes. The first rule for doing this is that their channel addresses must point to the same physical channel structure. This is not a problem in the linear address space of a transputer. Secondly, if hardware communication is being used, only 1 process should be using the channel; the transputer hardware does
not deal with 2 successive \texttt{in()}'s in a way that is consistent with channel usage.\footnote{The first \texttt{in()} would be treated as an \texttt{out()}. The functions \texttt{in()} and \texttt{out()} are machine instructions which implement message passing.}

On our host, a UNIX environment is used, and processes do not share the same address space. In this case, for example, a parent may pass a child the entire channel structure; however, thereafter, only 1 of these processes should be using the channel.

To date, programs have used only the channel graph set up at load time. However, channels may be dynamically created, destroyed and rerouted. This would require a high degree of cooperation amongst the processes comprising a program, and a lot of architectural details of the network would need to be known to the processes; for neither of these aspects would the current version of TMAP offer much help, nor does TMAP support dynamic switching of the crossbar links to better support a different topology of the communication channels. Section 9.2.3 discusses the topic of phases with respect to TMAP.

The blocking behaviour of channel communication is inherited from Trollius, which provides flexibility with optional buffering. Generally, at the network level of communication, the sender does not block, but the receiver does block. With hardware communication, both sender and receiver block, according to the transputer architecture.

In the occam language, a channel has the same blocking behaviour as hardware communication in TMAP. In occam, a channel is not a variable, and cannot be shared amongst different processes; there is no notion of a channel address. These restrictions place a heavier burden on the occam programmer, in the same way that assembly programming is more burdensome than even a moderate language such as C.
Chapter 7

Resource & Place Modules

The 3 modules Resource, Contract and Place do virtually all the work of choosing resources with which to implement the user's program; cf §4.4. The Prep-p module, Contract, is described in Appendix B.3.

7.1 Resource module

The Resource module reads the file SysInfo, described in Appendix C.1.1, and determines the number of transputers and ports that remain to be chosen; in that file, the user may have specified some of this hardware explicitly. The outstanding resource requirements, if any, are satisfied using the method of binary integer linear programming. The particular package that we use is XMIP83 [25]. The output of the Resource module is the file Resources, which is described in Appendix C.1.5.

The reader is referred to Chapter 2 for those architectural details of our system that this section assumes.
7.1.1 Problem formulation

There are 7 data ports available to users. Five of them are associated to the 5 components of our system, and are represented by the variables $c_0 \ldots c_4$, corresponding to components 0...4. The remaining 2 data ports are represented by variables $p_0$ and $p_1$, corresponding to ports 9 and 10.

Each of these 7 variables may take on only the values 0 and 1. Initially, a value of 0 for a variable indicates that the user has not explicitly requested that resource; a value of 1 indicates otherwise. After allocation, the values of these variables indicate which resources the user will have allocated for implementing the program.

Since this problem is small (only $128 = 2^7$ possible solutions), an exhaustive search for the "best" solution is reasonable (A precise formulation of "best" is given in §7.1.3). However, we give a more general approach by which larger problems may, also, be handled.

We will see that minimizing the cost function $7.8$ subject to the constraints $7.4$ solves our resource allocation problem.

7.1.2 Constraints

We define 3 functions of the 7 variables defined in the previous section.

\[
A_e = p_0 + p_1 \quad \text{(The number of extra ports allocated)}
\]

\[
A_t = 18c_0 + 18c_1 + 18c_2 + 19c_3 + c_4 \quad \text{(The number of transputers allocated)}
\]

\[
A_c = c_0 + c_1 + c_2 + c_3 + c_4 \quad \text{(The number of components allocated)}
\]
Let \( N_p \), \( N_t \) and \( N_c \), respectively, be the minimum number of ports, transputers and components requested by the user, as specified in the file SysInfo.

There are 3 constraints:

\[
A_c + A_e = c_0 + c_1 + c_2 + c_3 + c_4 + p_0 + p_1 \geq N_p
\]
\[
A_t = 18c_0 + 18c_1 + 18c_2 + 19c_3 + c_4 \geq N_t \tag{7.4}
\]
\[
A_c = c_0 + c_1 + c_2 + c_3 + c_4 \geq N_c
\]

7.1.3 Cost function

It is most important to minimize the number of components being used, \( A_c \), as this determines the number of concurrent users of our system. Next in importance is to minimize the number of transputers, \( A_t \), and finally, to minimize the number of extra ports, \( A_e \).

Rather than having 3 successive minimization problems, a single problem is constructed.

**Lemma 1** Let \( f \) and \( g \) be real valued functions on a finite set \( S \).

Then there is a constant, \( K \), such that if the function \( Kf + g \) has a minimum value at any point \( s \in S \), then \( f \) has a minimum at the same point \( s \).

The proof of this lemma consists of the derivation of Inequality 7.5, which gives a range for \( K \). The value \( K \) satisfies the lemma if

\[
(\forall s, t \in S) \ ( f(t) > f(s) \implies Kf(t) + g(t) > Kf(s) + g(s) )
\]
The right side of the above implication may be manipulated to give the following inequality (without the "max"), which gives a range of $K$ that satisfy the lemma.

$$K > \max_{f(t) > f(s)} \frac{g(s) - g(t)}{f(t) - f(s)}$$  \hfill (7.5)

The cost function is constructed by first choosing a constant $K_1$, according to the lemma, for the function

$$h = K_1 A_c + A_t.$$ \hfill (7.6)

Then choose a constant $K_2$ for the function

$$COST = K_2 h + A_e.$$ \hfill (7.7)

We will see in the next section that we may take $K_1 = 1$ and $K_2 = 3$, in which case, the cost function is

$$COST = 3(A_c + A_t) + A_e \quad = \quad 3(19c_0 + 19c_1 + 19c_2 + 20c_3 + 2c_4) + p_0 + p_1 \quad = \quad 57c_0 + 57c_1 + 57c_2 + 60c_3 + 6c_4 + p_0 + p_1.$$ \hfill (7.8)

### 7.1.4 The constants

Columns 2 and 3 of Table 7.3 show the minimum and maximum number of transputers in the number of components specified in column 1.

To calculate the constant $K_1$, we see from Table 7.3 that the largest value of the numerator in Inequality 7.5 is 0 (here, $f = A_c$ and $g = A_t$); so $K_1$ may be any value larger than 0, say 1, to keep the function $h$ of Equation 7.6 integral.

We apply the lemma to the function of Formula 7.7 to calculate $K_2$. Since the function $A_e$ takes on values between 0 and 2, the numerator in Inequality 7.5 is bounded
Table 7.3: For the constant $K_1$ of the cost function

<table>
<thead>
<tr>
<th>$A_c$</th>
<th>$\min A_t$</th>
<th>$\max A_t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>19</td>
</tr>
<tr>
<td>2</td>
<td>19</td>
<td>37</td>
</tr>
<tr>
<td>3</td>
<td>37</td>
<td>55</td>
</tr>
<tr>
<td>4</td>
<td>55</td>
<td>73</td>
</tr>
<tr>
<td>5</td>
<td>74</td>
<td>74</td>
</tr>
</tbody>
</table>

by $2$; so $K_2$ may be any number larger than $2$, say $3$. We have used the fact that the denominator of Inequality 7.5 is at least $1$, as it is an integer, being the difference of values of the function $h$.

**7.2 Place module**

The result of the Contract module is the graph of rnodes; cf §4.2. The Place module embeds this graph into the hardware specified in the file Resources.

This module is the weakest in terms of being user-friendly. This is not surprising, as the general mapping problem is intractable; the leading paragraphs of Chapter 1 discuss this briefly. Most TMAP sessions benefit from telling the system where to map some of the rnodes. An example of this is in the sample session of Chapter 5. However, we have not, to date, ever had to tell TMAP which link numbers of the transputers to use, although this could be done using a substitution on the file LkData, which is created by the Place module; cf Appendix A.1.

The actual embedding is done by adapting a greedy algorithm of Smedley [20], a former Masters student of A. Wagner; the latter is also the thesis advisor of the author,
and did the coding for the adaptation.

7.2.1 Outline

This section gives a brief description of the Place module.

First, RAM structures are initialized from the hardware database; cf §2.2. Next, the available hardware is obtained from the file Resources, and various heuristic parameters to guide the placement algorithm are calculated. These are briefly discussed in the description of the file HostMap in Appendix A.1, paragraph “SYSTEM FILES”.

The graph of nodes may have multiple edges between 2 given nodes. The first stage of the embedding algorithm begins by identifying multiedges as a single edge, and proceeds to embed this simpler graph into the hardware. An edge is mapped only if a direct hardware connection may be formed to carry the edge – dilations are not allowed. A dilation would occur if the edge would have to map to a path in the transputer network that contains transputers other than the 2 endpoints.

The second stage expands the mapped edges back to multiedges, and tries to further map these into the hardware, spreading multiedges over available links, where possible, and multiplexing over existing links, if necessary.
7.2.2 Discussion

This 2 stage approach was taken to increase the chance of avoiding unmapped edges (because of would-be dilations); the preference was to have multiplexing. However, examples have arisen where a "very" greedy algorithm would be more helpful i.e. to do the multiplexing on the fly. This tends to occur when the user specifies not to contract the graph of vnodes, and wants separate transputer links to implement multiple channels between processes.

TMAP does not currently support the option of doing the mapping in this "very" greedy way.

The handling of dilated edges is more in the domain of "routing" than "mapping", although the 2 problems are obviously closely linked. This version of TMAP does not do routing. Unmapped edges are routed at run-time using the Trollius router.
Chapter 8

Setup & Load Modules

The Setup and Load modules are the only modules which are specific to the low-level software being used, in this case, Trollius; cf §4.4. Appendix A.1 discusses all the files mentioned in this chapter.

8.1 Setup module

The primary purpose of the Setup module is to prepare for the Load module, stopping short of allocating resources and loading executables. This allows for the fastest repeated invocation of the Load module, assuming no changes are being made that require the attention of previous modules.

The file LkData is used to create the file bnail. This is the Trollius file which details, amongst other things, the transputers and ports being used, and their link-to-link interconnections. The Trollius numbering for the nodes is chosen here. In this implementation of TMAP, the Trollius number of a node is the same as the physical number; cf Appendix C.1.6. The bnail syntax does allow the user to choose a different numbering; perhaps it would be helpful to use the numbering of the rnodes (the output of the Contract module). This question was deemed unimportant for this first version.
of TMAP, as users are expected not to need to know about Trollius node numbers; in reality, they are helpful for debugging and accessing Trollius support routines e.g. state.

The file ProcMap is created from various previous files. Each line specifies the Trollius node number to which the executable of a vnode is to be mapped.

The file ChanLks is created; cf Appendix C.1.7. This file is the most helpful in determining, by inspection, whether TMAP has done a good enough job, with respect to dilations, in mapping the vgraph into the hardware.

8.2 Load module

The main job of the Load module is to patch the channel structures in each executable, before invoking the Trollius loader. The Load module accesses the files ProcMap, ChanLks, ChanVIds, ProcLst.sys, LoadLst and ParamLst in order to load the user's executables, in increasing order of the vnode number.

A future version of TMAP should allow the user to specify which executables to load, and the order in which to load them. Specifying the order would make some start-up logic for a program easier – a program can not claim links for a virtual circuit (cf Chapter 6) until they are no longer needed by Trollius for loading subsequent executables. Selective loading would allow a user, for example, to reload a process.
Chapter 9

Conclusions

The TMAP system is easy to use, both by the application programmer wishing to design, code and run a parallel program, and by the system developer wishing to enhance and expand the functionality offered by TMAP. The following two sections illustrate these properties. We emphasize here that a major improvement enjoyed by TMAP over Trollius is that application programs are topology-independent and, at the same time, they take advantage of efficient communication routines. Moreover, if the hardware topology changes, a program's source code need not be recompiled and no changes need be made to the Trollius router.

9.1 Experimental Results

The 2 sets of results described here are concerned with communication times within a ring. Comparisons are made of the times obtained when using the 3 different levels of communication; cf Appendix A.2. The basic functionality to achieve the apparent speed-up is already present in Trollius. The significance with respect to using TMAP is that the choice of communication level is easy to implement, both at run-time and when writing the application code; cf §4.3.
9.1.1 Spring

In this example, a round of communication consists, mostly, of all neighbours exchanging some information. The code was supplied by Russel Wvong. David Feldcamp added an X Windows interface, and I modified the communication calls to be used under TMAP. The ring has 18 nodes.

Table 9.4 shows the times for the ITB nodes to complete their message passing.

<table>
<thead>
<tr>
<th>ITB flag</th>
<th>Network</th>
<th>Virtual</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITB time</td>
<td>15.1</td>
<td>5.7</td>
<td>4.7</td>
</tr>
</tbody>
</table>

Table 9.4: Spring: ITB communication times (sec).

Table 9.5 was compiled using hardware communication amongst the ITB nodes, and shows the total program elapsed time, including sending the problem parameters to, and receiving the results from, the root transputer. The ITB-OTB communication is seen to be significant in this problem.

<table>
<thead>
<tr>
<th>OTB flag</th>
<th>Network</th>
<th>Virtual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total time</td>
<td>19.0</td>
<td>8.2</td>
</tr>
</tbody>
</table>

Table 9.5: Spring: Entire program times (sec).

9.1.2 Ring

This second example consists of messages being passed around a ring of 10 nodes. Each message is sent Count number of times and consists of Length number of bytes. This is
the example of §5. Table 9.6 shows some recorded times, including the communication with the host process, and the initial control message.

<table>
<thead>
<tr>
<th>Count</th>
<th>Length</th>
<th>Network</th>
<th>Virtual</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x10000</td>
<td>1.15</td>
<td>0.70</td>
<td>0.68</td>
</tr>
<tr>
<td>0x100</td>
<td>0x100</td>
<td>1.80</td>
<td>1.05</td>
<td>0.69</td>
</tr>
<tr>
<td>0x1000</td>
<td>1</td>
<td>18.8</td>
<td>6.65</td>
<td>0.85</td>
</tr>
</tbody>
</table>

Table 9.6: Ring: Entire program times (sec).

As expected, hardware message passing was fastest, with the most marked differences between the 3 types of communication occurring with many messages being sent – the overhead is more burdensome.

9.2 Future Work

9.2.1 TROLLIUS

In Prep-p, each procedure in the graph description file is limited to having at most 8 channels, and this restriction is inherited by TMAP. Thus, as alluded to in §1.1, each process “knows” only a small number of channel names viz. 8. Nonetheless, TMAP does allow a natural implementation of broadcasting, using the Trollius cast daemon. A broadcast occurs when a process sends a single message to a set of processes, usually covering several, if not all, nodes of the network.

A broadcast would be done by referencing a set of vnodes as the intended recipients. The vnode numbers are defined in the Graph Description File; cf §4.2. To implement this, each physical node of the network would run a TMAP broadcast process that would know
about the vnodes in the network. Each such broadcast process would have a channel, invisible to the user, that would be shared amongst the vnodes on that physical node. Additionally, the broadcast process would interact with the Trollius cast daemon.

9.2.2 LD-NET

As mentioned in §4.3, the implementation of TMAP may be modified to work with other low level systems, instead of Trollius e.g. Logical Systems’ ld-net. In this case, a process could be modeled as a set of object modules, or as a library. After contraction, the set of libraries mapping to a particular node would be linked into a single executable, as required by ld-net. This executable would also contain a router to implement network-wide message passing. The main() function in the executable would have the responsibility of creating the user’s processes and the router process(es).

9.2.3 High level

Bookkeeping

All the files mentioned in this section are discussed in Appendix A.1.

A complete TMAP session uses, at a minimum,

- 35 files in the main directory,
- 5 subdirectories, each containing its own files,
- an XX file for various procedures in the GDF, and
Chapter 9. Conclusions

- directories containing the user’s application code.

In addition, a session often uses

- substitution files created by the user to control the progress of the session (and copies of the standard files), and

- copies of the tm script, specific to particular configurations being used.

Needless to say, this is an overwhelming amount of data to keep track of – not for the system, but for the user who wants to feel that they are in control. A graphical interface is being developed to improve the ease with which information may be displayed for, and altered by, the user.

Phases

The current version of TMAP is static with respect to the communication graph. It is often the case during a program, that a communication pattern changes sufficiently and is constant over a long enough period, that a change is warranted in how TMAP supports the new message patterns.

At a software level, this is relatively easy to do – just define channel structures which point to the desired processes. The more difficult aspect is to reconfigure the hardware interconnections to efficiently support the new communication pattern.

A project is being discussed as to how to best implement phases in TMAP. It is hoped that Prep-p’s usage of phases may be largely applicable.
Bibliography


Appendix A

TMAP Implementation

The first 3 appendices consist of a collection of UNIX style manual pages, to flesh out the details of the TMAP implementation and operation. None of the Trollius manual pages are reproduced here, while some of the more relevant Prep-p pages are included.

In the text of the pages, a reference such as “foobar(l)” signifies that there is a manual page for foobar; however, if the information is not TMAP specific, the manual page will not appear in these appendices.

Appendix A.1 describes the user interface to TMAP, when invoking it from the shell prompt, while Appendix A.2 describes the interface that a programmer must use to take advantage of the communication facilities offered by TMAP.

A.1 TMAP

NAME

tmap - map processes and define a Trollius session.

SYNTAX


OPTIONS
-c  
only delete most output files of the specified modules. Do not do any processing. Substitutions are to be done.

-h  
print a help screen, along with the module names, in the order in which they are executed. Nothing else is to be done.

-m <module>  
execute only this module; this is equivalent to using the -f and -l options with the same module name specified. Module names are case insensitive.

-f <first>  
begin execution with the module having this name. The default is Vgraph, the 1st TMAP module.

-l <last>  
end execution with the module having this name. The default is Load, the last TMAP module.

-C n  
tell the Contract module not to do any contraction. This actually works by incrementing the number of nodes to which the module is allowed to contract, so the user should have specified at least the number of ITB vnodes that occur in the GDF. If more is specified, this flag has no effect. If fewer is specified, this flag may still be useful, considering how the OTB contraction is done; cf "OTB CONTRACTION", below.

-S n  
tell the Setup module not to make the Trollius route file. This might be preferred if the user anticipates substituting the route file for another. In any case, the Load module will make the route file if it is older than the bnail file.

-L [sn]  
send instructions to the Load module. The string parameter [sn] must not be empty i.e. either 's' or 'n' or both must appear. The 's' forces a spread(l) to be done. By default, if TMAP detects that a Trollius session is active, it will use that session to load executables
onto the network, and will not do a spread. If a Trollius session is not active, the 's' parameter has no effect. The 'n' prevents TMAP from running tm_pf, described in the section "ITB Printing", below. This only has an effect if TMAP is doing a spread. In this case, the user must run tm_pf in order to obtain the output from any tmprintf() calls occurring in the user's ITB code. Running tm_pf in a different window allows the ITB printing to be separated from the rest of the session.

[<GDF>] is the name of the Graph Description File. It is relevant only for the Vgraph module, and defaults to "GDF" if omitted.

[StdFile=SubstFile]
StdFile is expected to be one of the standard output files of the TMAP system. These files are listed in the next section, under the heading of its module. After completion of the module which produces StdFile, this file is copied to StdFile(STD), and the user's specified replacement SubstFile is copied to StdFile. Before the first specified module is executed, all substitutions of the previous modules are done. Any number of substitutions may be specified, separated by white space. There is no white space on either side of the '='.

DESCRIPTION
TMAP is an environment in which to design and run a parallel program on the transputer-based multicomputer at UBC. The user must choose a separate directory for each TMAP environment, as fixed file names are used. A complete TMAP session consists of the execution of the following 7 modules, in the specified order. Each module expects specific text files as input, and produces text files as output, to be used as input to subsequent modules. The substitution commands, described in the previous paragraph, allow the user to modify these files and fine tune the execution of TMAP. For each module, the output files, on which the substitution commands will work, are listed, below.
Vgraph
The user supplies a Graph Description File (GDF), defined in vgraph(1). The GDF describes the abstract processes that comprise the user's program; each such process is called a "virtual process", ViP or vnode, in accordance with the terminology of prep-p(1). ViP's communicate with one another using channels (called "ports" in the GDF).

ProcLst.sys -- specifies the procedure name of each vnode.

IOAdjLst.sys -- specifies the communication between vnodes.

The next 2 files reflect the changes to the above 2 files, after performing the OTB contraction, described in the section "OTB CONTRACTION", below.

ProcLst(1) --

IOAdjLst(1) --

ParamLst(1) --
specifies the command line parameters to be passed to each ViP.

callXX(1) --
compiles the XX files, which are described in the section "SPECIAL FILES", below.

IOLst(1) --
should be empty. It is used by the Contract module. In the prep-p system, this specifies external IO ports; this aspect of prep-p is not used in the TMAP system.

CHiPPparams(1) -- is expected by the prep-p modules.

ChanVIds(1) -- assigns a unique integer id to each channel.

LoadLst(1) -- specifies loading instructions for each ViP.
Weight

The 3 output files of the Weight module are produced by the XX(l) compiler, using as input only the .x files, there being 1 for each OTB-contracted procedure. The numbers in these files do not reflect the actual code that will be executed by the TMAP system at Load time. (The prep-p system does use this code to execute on its target architecture; the object output is supressed by TMAP.) These files are used to guide the modules Contract and Place, according to whatever heuristic is used. The user must decide just how much information is important to include in the .x source files.

CodeSize(l) --
specifies the number of bytes of code that would be generated for the target architecture of the prep-p system. The user may use this to tell TMAP the relative codesizes of the various OTB-contracted ViP's.

Degrees(l) --
specifies the degrees of the OTB-contracted ViP's. Channels between OTB processes are ignored until Load time.

EdgeWts(l) --
specifies the communication weights as seen by each end of each channel. Presumably, whatever goes in 1 end of a channel, comes out the other end. TMAP deals with this inconsistency by assigning a single weight to a channel, it being the sum of the 2 weights at the ends.

Resource

The resources requested by the user are specified in the file SysInfo(l). The Resource module compares these requests with the hardware database to come up with a minimal allocation of resources which satisfy all the requests of this TMAP session, and lists these suggested resources in the file Resources(l).

Cost.dat --

Cost.log --
Resources(1) --

Contract

VipToRp(1) --
This lists the intermediate contraction result produced by prep-p, and is used as input to produce the TMAP contraction, specified in RpLst, described below.

RpLst --
This is produced from VipToRp by moving OTBproc to be the sole member of a (perhaps new) solitary real node. If this is already the case, no change is made to the prep-p contraction. The syntax for RpLst is opposite to VipToRp -- each line of RpLst consists of 2 integer entries. The 1st is the ViP being referenced, and the 2nd is the real node onto which it is mapped.

MapInit --
This specifies the initial mapping of real nodes to host nodes. Each line of the file specifies the mapping of a real node to a physical node, and has the format:

<rnum> -> <pnum>

, where rnum is a real node, as defined in RpLst, and pnum is a physical node number. The host, logic, is numbered as -1.

Place

LkData(1) --

MapAll --
This file extends MapInit, described in the previous module. Each line of the file has the form

<rnode> --> <pnode>

, where rnode is a real node, as defined in RpLst, and
Appendix A. TMAP Implementation

pnode is the physical number of a node in the network, as described in LkData(1).

Setup

bnail. -- is a network configuration file to set up a Trollius session. The syntax is described in nail(1).

route. -- is produced by map(1) using bnail. as input. This file contains the routing tables that will be shipped to each transputer by Trollius while the system is being booted.

ProcMap -- specifies the Trollius node number of the transputer to which each Vip is mapped.

ChanLks(1) -- contains information about the non-local channels being used by the user’s program. A channel is non-local if its 2 ends are on different physical nodes of the network.

Load

star. --

This is not used by the TMAP system. It is produced by the UBC version of solder(1).

ITB PRINTING

The present version of Trollius, 2.0, has a bug with the file system, and stdio in particular. Two routines are supplied with TMAP to work around this.

int tmprintf(char*,...) is used like printf() except it is implemented with direct message passing to get its output to Logic, thus avoiding the filesys bug. This is available, also, OTB, but I suggest not using it there; in fact, I get system errors when using tmprintf() on Logic! I don’t know why. The return value is EOF for error, or the no. of characters output. The include file /project/transputer/include/tmap.h has the prototype for this function, and defines an upper bound for the length of output for a single invocation of tmprintf(), currently 1K
Appendix A. TMAP Implementation

This include file also defines tmprintf to be printf for an OTB compilation. So, if you really want to play around with tmprintf() on Logic, put #undef tmprintf at the top of the source file.

The routine which actually does the printing to the screen is called tm_pf, and is in /project/transputer/bin. So, choose a window where you want your ITB printing to appear, and run tm_pf&. When you do a tkill(1), this routine is killed, too, just like any other OTB process which does a kinit(1). For this reason, the Trillium kernel must be running on Logic for tm_pf to install itself.

tm_pf loops forever, receiving on a high-numbered event, currently 0x7ffffff5. When using tm_pf, don't use this event for receiving, otherwise, on Logic.

OTB CONTRACTION

The contraction routines that are used by TMAP are from the prep-p(1) system, and were not designed to deal with more than 1 CPU type. For this reason, the TMAP system requires that the user specify, in the GDF, whether a process is to run on the host, or on a transputer, as explained in vgraph(1). I do not see this as a serious restriction, as the OTB processes tend to be IO servers for the ITB processes, anyways.

The files ProcLst.sys and IOAdjLst.sys reflect the original GDF, whereas ProcLst and IOAdjLst reflect the OTB contraction. It is these latter 2 files which are used as input to the other 2 prep-p based modules, Weight and Contract.

The file ProcLst contains a phony procedure, OTBproc, which represents all the OTB processes defined in the GDF. The contraction output is adjusted so that OTBproc is not sharing a node with any other Vip. This is how the 2 CPU types are handled in the TMAP system.

The file IOAdjLst has 2 minor syntactic differences with IOAdjLst.sys, which make parsing easier. First, each line ends with a semicolon, and second, there is a space between
Appendix A. TMAP Implementation

each destination port name and the closing parenthesis. Finally, each port of OTBproc is of the form <name>_<N> , which means that port name is attached to ViP number N in the GDF, and as reflected in IOAdjLst.sys.

SPECIAL FILES
Certain files, once created, are never overwritten by the TMAP system. They are <GDF> , Makefile , bin/Makefile , tm , and all the XX files *.x . They are all created by the Vgraph module. The file SysInfo may be partially overwritten by the TMAP system, as explained in its own manual page.

SYSTEM FILES
The remaining files of the TMAP system are all system files -- the user is expected not to modify them, and need not be concerned with them. They are briefly described here, as an aid to understanding the workings of TMAP . The files are grouped according to the directory in which they exist.

bin/ In addition to the Makefile mentioned in the previous section, this directory contains the unpatched executable for each procedure in the GDF . As well, for each procedure, there is a TMAP symbol file having the extension .tym . This is used by the patch() routine in the Load module to find the correct offset in the executable, at which to write in the channel information.

The compiler tcc has fewer capabilities than cc or gcc . For this reason, the linking of ITB and OTB executables is handled differently.

OTBdir/ Each OTB procedure has a C source file here, in which the channel variables are defined. The compiled object files are here, too. The TMAP symbol file is created after linking, by searching the executable's symbol table for the channel array.

ITBdir/ This is the corresponding directory for the ITB procedures. Each source file, in addition, contains a directive for the assembler to pass the address of the channel
array up to the stage of linking the process' executable. At that point, the information is placed into the TMAP symbol file.

include/ Each procedure has an include file which declares the channels available to that procedure; these are here for the convenience of the user, to be included in the user's source code.

sys/ The files in this directory are either used only internally by TMAP, or are to be an aid in debugging. All of them, except for RawParamLst and OTBfg are created by the Place module.

RawParamLst --
This is created by the Vgraph module, and shows the parameters for each procedure, before variable substitution distinguishes the individual ViP's.

Guest --
The 1st line has 2 integer entries. The 1st is the number of guest nodes to be embedded into the available host nodes. The guest node numbering is the same as in RpLst. The 2nd entry is the number of edges in the guest graph. In this context, an edge corresponds to a non-local channel, as described for the file ChanLks. Each subsequent line is of the form:

g1 g2 weight

where g1 and g2 are 2 guest nodes, and weight is the communication weight, as described, above, for the file EdgeWts, of the edge connecting them.

HostMap --
The 1st line of this file consists of 2 integer entries. The 1st entry is the number of available host nodes. These are listed in the TRANS= line of the file Resources, described above. They are renumbered to begin at 0, and are augmented by 1 more node to represent Logic. The 2nd entry is the number of pre-mapped guest nodes. The 2nd line consists of 3 integer entries, which are used to guide the embedding heuris-
tic of the Place module. The 1st is the least no. of
guest nodes that should be mapped into the old box
(this is 1 of 2 boxes housing transputers in the UBC
network). The 2nd number is the total no. of avail-
able nodes in the old box. The 3rd entry is the no. of
wires between the old box and the new box, that may be
used by the embedding. There are as many remaining
lines as specified by the 2nd entry of the 1st line of
this file. Each line is a pair of integers. The 1st
is a guest node, and the 2nd is the host node to which
it is to be mapped. There should be at least 1 such
line, in order to ensure that the OTB procedures get
mapped to Logic! These lines are merely translations
of the information in MapInit.

swdata --
This consists of the crossbar settings that would
implement the connections specified by the Place module
(cf switch_set(1)).

Edges0 --
This file is produced by the algorithm which embeds the
guest graph into the host graph. The 1st entry is an
integer specifying the number of remaining lines, each
having the format

    index link1 link2

where index is the index of the edge, as listed in the
file Guest, and link1 and link2 are the link numbers
over which the channel (i.e. edge) is to be routed. If
link1 and link2 are -1, the edge must be either multi-
plexed over an existing edge, or dilated across inter-
mediate nodes.

Edges --
This is produced from Edges0 by doing the multiplex-
ing, using a sub-optimal knapsack approach with the
weights. Any edge having the links as -1 is neces-
sarily dilated for this embedding.

OTBfg --
This may be created by the Load module. When an OTB
process is specified to be run in the foreground (cf LoadLst(1)), the patched executable is copied from the bin directory to this file, and is loaded i.e. executed, last.

**USER CODE**

For each Procedure in the GDF, the user is expected to supply TMAP with the directory and file name of the code to be run on the network, as explained in the Makefile in the main directory. The user must have a Makefile in each such directory; there need be only 1 such directory, if all the user's code is compiled in that directory. The channel variables will be linked in by doing a make in the main directory of the TMAP session, so the user-supplied code is not expected to be fully linked. The ITB compiler tcc (man page lstcc(1)) and the OTB compiler cc(1) (or gcc(1)) have different capabilities; the format of the code that TMAP expects from the user is described separately for the OTB and ITB cases.

**OTB:** The default name for the code of procedure <name> is <name>.bin. The extension emphasizes that this is a partially linked executable. It may be produced by using 2 lines in the user's Makefile similar to:

```
<name>.bin: file1.o file2.o
   ld -r -o $@ file1.o file2.o -lg
```

The flag -lg should be used only in the case that the user is compiling with the debug -g option.

**ITB:** The default name for the code of procedure <name> is <name>.ta. This is an archive. The current version of tcc is not able to combine separate object files into a partially linked executable. It may be produced by using 3 lines in the user's Makefile similar to:

```
<name>.ta: file1.to file2.to
   rm $@
   trar cr $@ file1.to file2.to
```
A.2 Communication Routines

NAME
  ch_send, ch_recv - TMAP message passing

SYNOPSIS
  
  #include "tmap.h"

  int ch_send(TMCHAN *chan, void *msg, int msg_len, int flags);
  int ch_recv(TMCHAN *chan, void *msg, int msg_len, int flags);

  This manual page is modelled on the Trollius manual page for nsend/nrecv.

DESCRIPTION
  These 2 routines implement message passing in the TMAP system; cf tmap(1). One end of the channel sends, and the other receives the message.

  The chan parameter points to a TMAP channel structure, described in the next section. Such channels are defined in the Vgraph(1) module of the TMAP system. A user need only reference a channel variable; the object code which implements the channels is linked to the executable by TMAP.

  The user should include the header file which defines these channels, as described in tmap(1), section "SYSTEM FILES", paragraph "include/". A channel is said to be local if both ends are on the same physical node. This property of a channel depends on the mapping of the user's processes into the network.

  The msg parameter points to the location in RAM of the 1st byte of the buffer to be used for sending/receiving.

  The msg_len parameter specifies the no. of bytes of data to be sent/received, starting at the location msg. It is recommended that the sender and receiver specify the same message lengths; otherwise, the most efficient level of communication, at the hardware level, may not be used. The Trollius routines, which are used for higher level message passing, do allow length mismatching. The user may use a more complicated strategy at the hardware level by using
multiple sends or receives to achieve matching lengths.

The flags parameter specifies how the message is to be sent/received, and are all from Trollius, except for THOLD. Their functions fall into 3 categories:

1) Buffers
   BYBUF -- Don't use buffers on this node.
   NOBUF -- Don't use buffers anywhere along the route.

2) Data translation
   This is described in the section "DATA REPRESENTATION", below.

3) Message level
   KHOLD -- Use a Trollius virtual circuit.
   THOLD -- Use hardware communication.
   By default, both routines use the Trollius routines dsend(1) and drecv(1), unless a hardware virtual circuit is specified by the user. A virtual circuit (VC) comes in 2 flavours -- using Trollius, and using hardware. This version of TMAP implements hardware communication only on ITB nodes, in which case, Trollius is dispensed with, and simple in/out pairs are used.

The current version of TMAP has restrictions on the use of VC's. A local OTB channel should not use a VC at all. If either end of a channel is OTB, do not use a hardware VC; even if a Trollius VC is being used, beware that this may interfere with other uses of the port e.g. printing. If both ends of a channel are ITB, both types of VC may be used, and the hardware version is recommended. Actually, for a local ITB channel, a VC request is always implemented as the hardware type. It is recommended that the Trollius type of VC be used only for communicating with an OTB process.

THE TMAP MESSAGE STRUCTURE
   Both functions accept a pointer to a channel structure which is defined in "tmap.h" as:

   typedef struct
   {

int node; /* Initially, the node number at the other end. */
int id; /* The channel id at that node (plus other info). */
}
TMCHAN;

The node field is patched at load time to be the Trollius node number at the other end of the channel.

The id field encodes the Trollius event to use in non-VC message passing, as well as whether or not the channel is local, and whether or not the channel is initialized. A local channel is initialized if the node field either is the channel synchronization word for hardware message passing, or it points to the node field at the other end, which is the synchronization word. In this version of TMAP, a local OTB channel is never initialized. A non-local channel is initialized if the id also encodes the hardware link to be used for the sending and receiving of messages.

DATA REPRESENTATION

On nodes of different types, data may have different representations. For example, integers may be stored with the most significant byte first in memory (big-endian) or with the most significant byte last in memory (little-endian).

The flags parameter of the sender can be set to the following data representation flags (they will have no effect when used with ch_recv). Each flag assumes a data type, and will make the appropriate change in the data representation of the message. They will have no effect if data conversion is not needed.

DINTMSG msg points to integers.

DFLTMSG msg points to single precision real numbers.

DDBLMSG msg points to double precision real numbers.

DRAWMSG msg representation will not be changed.

If msg contains a mixture of data types, the user will have to change the representation using the functions ltot(l),
BLOCKING
The blocking behaviour is the same as for the Trollius routines nsend/nrecv, except if the THOLD flag is set, which is allowed only for ITB-ITB communication. In this case, either end will block until it has sent or received its specified length of message.

RETURN VALUE
Upon successful completion, 0 is returned. Otherwise, -1 is returned and the global variable errno is set to indicate the error.

ERRORS
The following error codes are in addition to those returned by Trollius.

[ELINKUSED] ch_send or ch_recv failed because the link required to set up a virtual circuit is already being used by another channel.

[ENOVEVENTS] The Trollius kernel ran out of virtual event descriptors.

[ECHANLOCAL] Trollius reports that an uninitialzied non-local channel is actually local. Unless the user is creating channels at run-time, this indicates an error in the TMAP loader.

BUGS
Hardware VC’s involving a OTB process are not implemented. The routines corresponding to Trollius’ dtry_send/dtry_recv are not implemented.
Appendix B

Prep-p Based Modules

The manual pages in this appendix discuss the 3 modules of Prep-p that were modified for use in the TMAP system. The manual pages are slightly altered to reflect these changes.

B.1 Vgraph Module

NAME

GraphCom - Prep-P graph compiler

SYNOPSIS

GraphCom

DESCRIPTION

This man page is based on the original prep-p man page for GraphCom(l), and describes the modified version which is used for the TMAP(l) system at UBC. For the sake of clarity, the differences with the original GraphCom are not described, here.

GraphCom has two functions: (1) it serves as the first module of the prep-p mapping preprocessor, and (2) it is used to define library graph descriptions for later use. Which of these is done depends on the first word in the input file (standard input). If this is the word "library", then function (2) is assumed. If it is anything else, it is interpreted as a graph type (e.g. tree, line, irregular),
and function (1) is assumed.

As its main function (1 above), GraphCom reads a graph description from standard input and produces the standard output files IOAdjLst, ProcLst, IOAdjLst.sys, ProcLst.sys, IOLst, ParamLst, ChanVIds, LoadLst, the shell script callXX, the Poker file CHiPParams, and an XX(1) file corresponding to each procedure name in the GDF, for use in the Weight(1) module. If a particular XX file already exists, it is left alone. In this way, the system produces skeleton XX files which the user may modify. These files are essentially an expansion of the graph description given by the user. Their format is documented in separate man pages. Briefly, IOAdjLst(1) is an adjacency list, ProcLst(1) associates process code names with virtual process numbers, IOLst(1) gives details of external I/O streams, files and columns (indices) in files. CallXX(1) is a shell script which compiles the process XX(1) source code required. This XX code is used in the TMAP system only to produce the output of the Weight module, described in weight(1) and tmap(1). The actual executable code provided by the user is probably written in C. The user tells TMAP where this code is using the file Makefile, which is one of the non-standard files of the TMAP system, and is described in tmap(1).

GRAPH DESCRIPTION LANGUAGE

For a complete explanation of the graph description language, consult the document "A Child’s Garden of GraphCom" by Dwight Newton. The following is just a brief overview, which does not attempt to describe all the features available. In particular, the library definition facility is not described here.

A graph description contains 3 main parts: a graph type, a global bind list, and a number of procedure declarations.

The graph type specification serves two functions. First, the connectivity is checked by subsequent modules for some graph types. Second, the graph type determines which contraction and placement strategies will be used for the mapping of the parallel program described in the graph description. The graph types known at the time of this writing are
Appendix B. Prep-p Based Modules

line, loop, mesh (square mesh), hexmesh, torus, felem (finite element), bincube, shuffex, s4pin (4-pin shuffle), tree, treefold, butterfly, ccc (cube connected cycles), cyclic (cyclic contraction), even (even contraction), irregular (probabalistic or graph traversal contraction).

The global bind list gives the values (as C-language expressions) for symbolic names used in the procedure declarations. The expressions can be in terms of other symbolic names defined later or earlier, and in terms of a node number i. The most important symbolic names are "nodemin", "nodecount", "OTBmin", "TRANScount" and "PORTcount".

There are nodecount procedures numbered consecutavely from nodemin, a non-negative integer; these 2 constants must be specified explicitly. All the procedures numbered from OTBmin upwards represent processes that will be run on the host, logic. All other processes will be mapped to a transputer. This distinction is made to facilitate the contraction of the processes onto the available hardware. The default value for OTBmin is -1, an invalid index, which specifies that there are no processes to run on the host. TRANScount is the minimum number of tranputers onto which the processes will be contracted; if, during resource allocation, more transputers happen to be available, the user may specify, in the file SysInfo, whether to allow the Contract module to use all the available transputers (TRANSfill = 1), or to restrict the contraction to at most TRANScount transputers (TRANSfill = 0). The default value for TRANScount is 18, the size of components 0, 1 and 2 in the transputer system. Similarly, PORTcount is the minimum number of ports (hardware links with logic) that the user wishes to obtain from the Resource module. The PORTfill variable in SysInfo specifies whether to use all available ports during the Place module, or just the number PORTcount. The default value for PORTcount is 1.

An example of a global bind list is:

{ 
k = 5;
nodemin = 0;
nodecount = n;
Appendix B.  Prep-p Based Modules

OTBmin = n-1;  /* The last 2 processes will run on the host.  
TRANScount and PORTcount revert to their default values, described above. */
n = 1<<k;  /* 2**k */
}

which could be used when the number of nodes and the log (base 2) of that number are used later in the procedure declarations. It is generally preferable to use symbolic names to simplify the expressions in the procedure declarations (see below), since it improves readability of the description. Further, for each of the values of i (there are exactly nodecount of them), each symbolic name is evaluated at most once, no matter how many times it is used. If common subexpressions are given one symbolic name, considerable running time can be saved.

The procedure declarations specify the name of the source code files to be used for the process, and the connectivity of the processes. A procedure declaration can be a use-clause which imports an entire predefined subgraph from a library definition, or it can be a simple procedure declaration for just one procedure. This is an example of a simple procedure declaration:

procedure LEAF{ i , "I am a leaf" }
    nodetype : { i >= ((nodecount+1)/2 && i < nodecount }
    port PARENT : { i/2 }
    when {leaf_io} port OUTPORT : output dataout

A simple declaration specifies a procedure name, 2 parameters to be passed at load time, a condition on the node number i under which the procedure runs as process i, and the ports through the procedure communicates to other processes. The parameters are not separated by commas. The condition is an expression which contains C-operators, symbolic names defined in the global bind list, and the node number variable i. If the condition evaluates to a non-zero value for some particular i, then the source code for process i is taken from the XX file which has the procedure's name (extended by ".x") in the current directory. The example above would generate code from the file LEAF.x for the leafs of a full binary tree with 1-based numbering.
The port declarations contain the port name, the destination of the port, and possibly a guard expression if the port is used only under certain conditions. The destination is either a node number (if the port leads to another process), or an external I/O stream/file name. Both the destination expression and the guard expression can contain symbolic names (from the global bind list), C-operators, and the node number variable i.

A use-clause contains a file name (i.e. where to read the library definition from), and a local bind list. The following is an example of a use-clause:

```
use "/usr/local/prep-p/libdefs/tree"
{
  k = ncl;
  nodemin = nm1;
}
```

where ncl and nm1 would be defined in the global bind list. What this says is that you want whatever the tree-library defines included in your graph, and that you want to replace the library's definition of what nodemin and k by ncl and nm1, respectively. Note that changing nodemin (or k) in this local bind list does not affect what they are in the global bind list, or in other local bind lists. Thus, you can include different libraries, or the same library several times, each time with different values for the symbolic names.

**AUTHOR**

Darin Johnson wrote the original version. Dwight Newton completely revised the program. Bernd Stramm wrote this man page. Norman Goldstein modified GraphCom for use by the TMAP system at UBC.
Appendix B. Prep-p Based Modules

B.2 Weight Module

NAME

callXX - Is a C Shellscript which calls the XX compiler to compile all procedures indicated in an algorithm's graph description.

DESCRIPTION

The UBC TMAP system uses a modification of the original prep-p callXX(l). The assembler output files (.s) are suppressed. Parameters to the XX(l) procedures have no effect. Instead, parameters are passed to the procedures in the original Graph Description File, as described in vgraph(l).

The XX language is a simplified sequential programming language for defining the codes to be executed by the processing elements of the CHiP computer. The XX compiler is used to compile the program written in XX language and to find all the possible syntactic errors.

In order to get a better contraction result, the contraction algorithm in Prep-P uses two additional pieces of information on the given graphs. First, the EdgeWts shows the port usage in each process of an algorithm and the maximum degree of all the processes. Secondly, The CodeSize shows the actual codesize of a given code name. The XX compiler has been modified by installing two separate passes to compute these two files.

The callXX script, created by the GraphCom module, consists of the command: XX <codename1> <codename2> ... <codenamen> where <codename1> through <codenamen> are exactly those Dos Equis procedures which will be executed by Virtual Processors of the algorithm.

The usual name for the <codenamei> procedures are ViP0.x, ViP1.x, etc.

In addition, the callXX script checks that all procedures passed without error through the XX compiler, and sends an error message to the user if this is not the case.
FILE
Output Files:

./EdgeWts - the list of heavy and light edges, and the weights of the light edges.

./CodeSize - the codesize for a given code name. This is the no. of bytes of Intel 8051 code that would be produced for the Poker system, and may be used as a relative estimate of the code size for any processor, as reflected in the XX source code, NOT in the source code that will eventually be loaded onto the transputer network by the TMAP system.

./Degrees - the number of channels possessed by each ViP.

AUTHOR
Susan Merritt. Shin-Tih. Modifications for the TMAP system at UBC were made by Norman Goldstein.

B.3 Contract Module

The minor change for TMAP is described in A.1, paragraph “OTB CONTRACTION”.

NAME
contract - partition virtual processes into groups of real processes

SYNOPSIS
contract [-D] [-q] [-o][-r]

DESCRIPTION
Contract is the module of prep-p which partitions a group of virtual processes (vips) onto a smaller group of real processes (reps). Contract runs after GraphCom and callXX in the execution of prep-p. It takes the files generated by GraphCom from the graph description file, and generates new files to reflect the mapping. These files are then later used by placement.
Contraction first determines whether a given graph needs to be contracted (i.e. the number of vips is greater than the number of reps). If so, it performs the contraction algorithm. If not, it just normalizes the node numbers for vips so that they are contiguous, generates the output files for a one to one mapping, and returns.

The contraction algorithm acts as follows: if the graph is a library graph for which a library contraction exists, this template will be used to generate the partitioning. If the graph is irregular in form (i.e. has no library contraction), contract uses a local neighborhood search (LNS) algorithm to determine a good configuration. LNS sets up an initial partitioning configuration, and then iteratively selects a vip at random, moves it from one rep to another, and compares the two configurations via a cost function. If the cost is lower for the new configuration, it is retained. Otherwise the older configuration is restored. The process ends when a threshold of repeating failures occurs.

Library contractions are defined procedurally, i.e. if the user wishes to add a library contraction, it must be hard coded.

For the LNS contraction, there are three cost functions available. The original cost function attempts to minimize the edges between reps, while attempting to evenly distribute vips among the available reps.

In the second cost function, we differentiate between heavy and light edges. Light edges have a weight associated with them, which denote the predetermined amount of traffic over the edge; heavy edges are assumed to bear a large amount of traffic, since the amount is not determinable. In this calculation, external edges to I/O pads are handled in the same fashion as internal I/O edges between vips. This cost function works using a three tiered group of functions. If the two configurations are equivalent for function A, function B is used, and likewise for function C. All three functions attempt to yield an even distribution of vips on reps, but also attempt to control the distribution of edges between vips/reps. Function A attempts to maximize the
number of heavy edges within a given rep; function B does the same for light edges, but attempts to maximize the sum of weights of the light edges within a given rep; and function C attempts to minimize the number of edges between reps.

The third cost function is available only to prove the validity of the first two. It randomly configures the graph, attempting only to ensure that the graph satisfies certain constraints (see below).

The -D flag causes a rather large amount of unintelligible gibberish to be printed to the screen; this is not highly recommended for the casual user. Normally, contract gives some information as to what it is doing, and prints out a summary of the results of its work. If the -q flag is used, this output will be silenced.

FILES

Input Files:

./EdgeWts - the list of heavy and light edges, and the weights of the light edges
./ProcLst - the name of the executable code for a given vip
./CodeSize - the actual codesize for a given code name
./IOAdjLst - a list of the interconnectivity between vips

Output Files:

./ConPadMap - the mapping of virtual pads to real pads
./ConAdjLst - the mapping of virtual adjacency lists onto real adjacency lists
./VipToRp - the mapping of vips to reps

AUTHOR

Bernd Stramm
Ken Cooper
Appendix C

File Descriptions

The manual pages in this appendix discuss the formats of some of the files used in the TMAP system. The first section consists of files that are unique to TMAP. The second section consists of Prep-p files which are used by TMAP as well.

C.1 TMAP specific files

C.1.1 SysInfo File

NAME

SysInfo - system information for a TMAP session.

DESCRIPTION

SysInfo is produced by the 1st module, Vgraph, and is used by most subsequent modules. The file consists of 11 fields, broken up into 4 blocks. The 1st 2 blocks are overwritten by the Vgraph module, using whatever is specified in the Graph Description File (GDF), while the last 2 blocks are only ever changed when the user explicitly edits SysInfo.

The 1st block summarizes information which should not be changed by the user, except by rerunning the Vgraph(1) module, and has the format:

\[
\text{PROCcount} = \langle\text{PROCcount}\rangle ; \\
\text{nodemin} = \langle\text{nodemin}\rangle ;
\]
nullcount = <nullcount> ;
OTBmin = <OTBmin> ;

where PROCcount is the number of procedures defined in the
GDF, nodemin is the smallest index of a ViP, nodecount is
the number of ViP's in the GDF, and OTBmin is the smallest
index of an OTB ViP (cf vgraph(1)); it is -1 if there are
no OTB procedures. Both PROCcount and nodecount reflect the
values before OTB contraction (cf tmap(1)).

The 2nd block summarizes the resource requests that may be
made from the GDF, and has the format:

TRANScount = <TRANScount> ;
PORTcount = <PORTcount> ;

where TRANScount is the least no. of transputers that the
user will have allocated, and PORTcount is the least no. of
ports that the user will have allocated.

The 3rd block allows the user to specify which ports and
which components of the network to use (Individual tran­
sputers may be specified using the file Resources(1)). The
format is:

COMPcount = <COMPcount> ;
Components = COMPlist ;
Ports = PORTlist ;

where COMPcount is the least no. of components that the user
will have allocated, and COMPlist and PORTlist are lists of
the specific components and ports to use. The syntax is
illustrated in the EXAMPLE, below.

The 4th, and last, block controls the usage of the allocated
resources. The format is:

TRANSfill = <TRANSfill> ;
PORTfill = <PORTfill> ;

where TRANSfill and PORTfill are boolean variables that
specify whether to use extra resources that may be the
result of resource allocation; for example, transputers are
allocated in blocks, each block being a reset component. A value of 0 means not to use more than is specified in the corresponding "count" variable. The default value for TRANSfill is 0, while the default value for PORTfill is 1.

**EXAMPLE**

Here is a sample SysInfo file.

```plaintext
PROCcount = 3 ;
nodemin = 1 ;
nodecount = 7 ;
OTBmin = 6 ;

TRANScount = 20 ;
PORTcount = 2 ;

COMPcount = 1 ;
Components = <3> ;
Ports = <0> <-1> ;

TRANSfill = 0 ;
PORTfill = 1 ;
```

The user wishes to have at least 20 transputers and 2 ports. One component is specified viz. number 3. Also, 2 ports are requested, but only 1 is specified viz. number 0; the `<-1>` is used to terminate an under-specified list. Presumably, the components allocated would consist of components 0 and 3, having a total of 37 transputers. Since TRANSfill is 0, at most 20 of these would be used. Had the user not specified any particular components, the Resource module would have allocated components 0 and 4, possessing exactly 20 transputers and 2 ports. Users should refrain from restricting the choice of resources, unless there are good reasons to do otherwise.

### C.1.2 ParamLst File

**NAME**

ParamLst- command line parameters.
Appendix C. File Descriptions

DESCRIPTION

ParamLst is the file which specifies the command line parameters which are to be passed to each executable at load time. This file is produced by the Vgraph module of the TMAP system, and consists of repeated lines of the following format:

\[ \text{vnum} : \text{count} : \text{ParameterList} \]

where vnum is the number of the ViP being referred to, count is the number of parameters being passed to this process (count is 0 if no parameters are being supplied, although, in a C program, argc is 1), and ParameterList is a sequence of count strings, each having the same format as described in LoadLst(1) for the instructions to the loader.

The ViP numbers are listed in increasing order.

C.1.3 LoadLst File

NAME

LoadLst- loader instructions.

DESCRIPTION

LoadLst is the file which instructs the Load module of the TMAP(1) system how to load the executables onto the nodes of a network. This file is produced by the Vgraph module, and consists of repeated lines of the following format:

\[ : \text{vnum} <\text{instructions}> \]

where vnum is the number of the ViP being referred to, and instructions is a text string to be interpreted by the loader. In this string, a pair of double quotes ("\) may be used to hide white space. The back slash (\) may be used to introduce a literal (") or a literal (\). The meta string must not be empty. The empty instruction should be written as "" , which is, in fact, the default instruction for each process.

For ITB loading, consult loadgo(1) for the allowable
Appendix C. File Descriptions

options.

The only allowable option for OTB is "f". This instructs the Load module of TMAP to put that process in the foreground; the default is to run each executable in the background using the ampersand (&). It is an error to specify more than 1 process as foreground, as they would compete for stdin.

The order of loading is as specified in this file, with 1 exception. If any OTB process is specified as foreground, it is saved till the last. This is done as a convenience (to the design of TMAP). A user's program should be designed so that the order of loading is not significant.

: 3 "-o OUT3"

Presumably, vnode 3 is an ITB process. Its stdout is to be redirected to the file OUT3.

: 5 "f"

Again, presumably, vnode is an OTB process, and it will be run in foreground.

C.1.4 ChanVIds File

NAME

ChanVIds - channel virtual ID's.

DESCRIPTION

ChanVIds is the file containing the TMAP encoding of the channels which are specified in the Graph Description File. This file is produced by the Vgraph module, and is used by the Load module to patch the correct information into the executables on their way to their destination nodes. The file consists of repeated lines of the following format:

: vnum ChannelList ;

where ChannelList specifies all the channels connecting with
the ViP number vnum. ChannelList is a sequence of entries of the form

: name = { vnum_dest , id_dest }

where name is the name of the channel, as known to ViP vnum, vnum_dest is the number of the ViP at the other end of the channel, and id_dest is the id which is used by ViP vnum to send along the channel. The id's are distributed in such a way, that ViP vnum receives along the channel by using the id obtained by toggling the lowest order bit of id_dest. It is necessary to assign a different id to each direction of the channel, since both ends of the channel might be loaded onto the same physical node; this ensures that a receiver obtains only what the other end has sent.

C.1.5 Resources File

NAME

Resources - resources to be used by a TMAP session.

DESCRIPTION

The file Resources is produced by the Resource module, and completely lists all resources available to the Contract and Place modules.

The 1st line has the form

Error= <Error>

where the integer value Error should be 0.

The next 3 lines specify how many transputers, ports and components will be used. It has the format

TRANSplace= <TRANSplace>
PORTplace= <PORTplace>
Using <num_comps> components.

where TRANSplace and PORTplace are the numbers of transputers and ports that the Place modules should use to
actually implement the user’s program, and num_comps is the
number of components allocated.

The next 2 lines specify the components and ports, as in the
file SysInfo(1). The format is

Components: COMPlist ;

Ports: PORTlist ;

Where COMPlist and PORTlist have the same format as in the
file SysInfo(1). Of course, neither list should now be
under-specified.

The next field specifies the transputers to be used by the
Place module. The format is

TRANS= TRANSlist ;

where TRANSlist lists the physical numbers of transputers
that the Place module may choose from. The EXAMPLE, below,
illustrates the format.

The final field specifies the inter-crossbar wires that the
Place module may choose from. The format is

XXWIRE= <inter-box-wires>
   <other-wires>;

where inter-box-wires lists the available wires connecting
the 2 boxes housing the tranputers, and other-wires lists
all the other inter-crossbar wires. Again, the EXAMPLE,
below, has an illustration

EXAMPLE

Here is a sample Resources file.

Error= 0

TRANSplace= 10
PORTplace= 2
Using 2 components.
Components: <0> <4>; 

Ports: <0> <6>;

TRANS= 0 73 1 24 9 17 21 20 16 12 13 19 22 14 23 15 11 10 18;

XXWIRE= 2 9
 0 1 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25;

The system will be using only 10 tranputers, but the user has requested that 2 ports be used. As a result, components 0 and 4 have been allocated. The tranputers in these 2 components are listed in the "TRANS=" field. The ordering of the tranputers in this list is significant to the heuristics of the Place module. Generally, the most "useful" tranputers are listed first. As seen in the 1st line of the "XXWIRE=" field, only 2 of the inter-box wires are allocated to this session. The next section explains this policy.

INTER-XBAR WIRES

The routine xxwires lists some hardware information, including the numbering of the inter-xbar wires. The wires are numbered upwards from 0. For each such index, a wire entry has the format

index=( xbar1,link1 ; xbar2,link2)

where index is the wire number, xbar1,link1 is 1 end of the wire, and xbar2,link2 is the other end of the wire.

There are only 8 inter-box wires. To avoid conflicts at load time, each of components 0,1,2 and 3 allows the user to use 2 of these wires, as listed in the following table

<table>
<thead>
<tr>
<th>Component</th>
<th>Inter-box wires</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2,9</td>
</tr>
<tr>
<td>1</td>
<td>3,8</td>
</tr>
<tr>
<td>2</td>
<td>4,7</td>
</tr>
<tr>
<td>3</td>
<td>5,6</td>
</tr>
</tbody>
</table>

The policy is, that if a user has not been allocated a com-
ponent, then the associated inter-box wires should not be used.

C.1.6 LkData File

NAME
LkData - link interconnection specifications.

DESCRIPTION
LkData is produced by the Place module of the tmap(l) system. It has the format for specifying link interconnections in the transputer network at UBC, and consists of repeated lines of 4 integers:

node1 link1 node2 link2

where node1 and node2 are 2 physical node numbers in the network. Currently, transputers are numbered from 0 to 73, and the host, logic, is -1. Transputer link numbers range from 0 to 3, and host link numbers range from 0 to 10. The line specifies that link1 on node node1 is to be connected to link2 on node node2, in the sense of being either hardwired, or by using up to 2 crossbars to implement the connection. As far as message-passing software is concerned, this is a direct physical connection. In reality, there is a delay incurred for each C004 crossbar used to effect the connection, of around 18%. This would give a 50% reduction in throughput if 3 crossbars were to be used.

C.1.7 ChanLks File

NAME
ChanLks - channel initial links.

DESCRIPTION
ChanLks is produced by the Setup module of the tmap(l) system, and describes the initial routing of messages along each non-local channel (A channel is non-local if its 2 ends
Appendix C. File Descriptions

are on different physical nodes. This file is used by the Load module to patch the correct information into the executables on their way to their destination nodes. The file consists of repeated lines of the following format:

\[ <\text{vnode1}> (<\text{link1} \text{name1} >, <\text{name2} \text{link2}>) \text{vnode2} \]

where the integers vnode1 and vnode2 are the ViP numbers at either end of the channel, the strings name1 and name2 are the names by which the channel is known at the respective ends, and the integers link1 and link2 are the links by which messages are received and sent at the respective ends of the channel. A value of -1 signifies that the channel is dilated i.e. it does not connect 2 processes on nodes having a physical connection. This version of TMAP does not deal with the routing of dilated channels.

C.2 Prep-p files

C.2.1 ProcLst File

NAME
ProcLst - Virtual processor to Procedure name mapping file.

DESCRIPTION
ProcLst is the file used by the Prep-P system to map virtual processors (nodes in the original computation graph) to procedure names (name of the code to be executed by that virtual processor).

The file consists of repeated lines of the following format:

\[ : N : \text{procedure-name} \]

where \( N \) is the number of the virtual processor and procedure-name is the name of the XX routine to be executed by \( N \). If an algorithm uses a graph generation program (such as genccc.c), then the XX code generated for execution by virtual processor \( N \) is usually called ViPN.x
The ProcLst file is produced by the GraphCom routine. It is used by the genCN routine to generate the CodeNames file for Poker and by Premux.

AUTHOR
Susan Merritt

C.2.2 IOAdjLst File

NAME
IOAdjLst - detailed adjacency list file for Prep-P graphs

DESCRIPTION
IOAdjLst is the file used by the Prep-P system to describe the adjacencies of the virtual processors (ViPs) in a graph, as well as the ports which connect the adjacent ViPs or extern I/O pads.

The file consists of repeated lines of the following format:

: V : (pname, pname) V1 : (pname, <) -m : (pname, pname) V2 : (pname, >) -n : . . .

where V is the number of an ViP, the Vi are numbers of ViP adjacent to it, negative number -m or -n represents the i/o pad adjacent to (instead of another ViP), and the values in parentheses are the portnames on V and on Vi which form the two ends of the connection. Note that if V is connected to an i/o stream, we use the symbol "<" or ">" to distinguish the input port and the output port.

IOAdjLst is generated by the Prep-P program GraphCom. The file is used as input for later stages of the Prep-P system such as contr and genPN.

AUTHOR
Shin-Tih Jing
Appendix C. File Descriptions

C.2.3 CodeSize File

NAME
CodeSize - file containing the size of Intel 8051 code for each procedure node

DESCRIPTION
CodeSize is the file used by the Prep-P system to evaluate contraction. The file consists of repeated lines of the following format:

procedure-node-name : S

where procedure-node-name corresponds to a XX file and S is its size of Intel 8051 code.

This file is produced by the XX compiler and is used by the contraction routine, amount other files, to generate ConPadMap, ConAdjLst and VipToRp.

C.2.4 Degrees File

NAME
Degrees - the degree of each ViP.

DESCRIPTION
The file Degrees just lists the number of ports connected to each ViP in a TMAP session. This file is produced by the Weight module, and consists of 1 integer entry for each ViP, listed in order of increasing ViP number.

C.2.5 EdgeWts File

NAME
EdgeWts - file containing internal and external edges weight of each procedure nodes

DESCRIPTION
Appendix C. File Descriptions

EdgeWts is the file used by the Prep-P system to evaluate contraction.

The first line of file indicates the number of procedure nodes. The rest of the file consists of repeated lines of the following format:

procedure-node-name : port-name weight | [port-name weight | ]

where procedure-node-name corresponds to a XX file, port-name is one of the defined port of the node, and weight is an assigned value predicting the amount of traffic through this port.

This file is produced by the XX compiler and is used by the contraction routine, among other files, to generate ConAdjLst, VipToRp and ConPadMap.

C.2.6 VipToRp File

NAME

VipToRp - Virtual Processor to Real Processor mapping file

DESCRIPTION

VipToRp is the file used by the Prep-P system to map virtual processors (nodes in the original computation graph) to real processors (nodes in the contracted graph).

The file consists of repeated lines of the following format:

: R : V1 V2 V3 . . .

where R is the number of a real processor, and the Vn are numbers of virtual processors mapped to it.

The VipToRp file is produced by the contraction module.

AUTHOR

Susan Merritt, N. Goldstein (minor deletion for TMAP).
Appendix D

Sample Source Files

This appendix contains the source code for the sample program of Chapter 5. As mentioned there, each message passing experiment is preceded by the passing of a control packet around the ring.

D.1 Control Information

The 3 fields of the control structure tell each node what it needs to know about the upcoming rounds of message passing.

/* Header file control.h for the sample TMAP program. */

struct control
{
    int count, /* The number of times to send the message. */
    length, /* The number of bytes in each message. */
    type; /* Either 0, KHOLD or THOLD. */
};
D.2 Host Code

The host process performs an endless loop, setting up the message passing experiment as directed by the user, and printing the time to complete the experiment. The host and the transputer have different internal representations for storing integers in RAM, but by using the \texttt{DINTMSG} flag, Trollius does the conversion, and this is otherwise invisible to the user.

/* Source file Host.c for the sample TMAP program. */

#include <stdio.h>
#include <sys/time.h> /* For time functions. */
#include "Host.h"   /* The vnode include file. */
#include "control.h" /* Specific to this example. */

/* This is the OTB IO server for the example. 

The prompt is for 3 values: 
count (Hex integer) -- The number of times to pass the packet around the ring. 
length (Hex integer) -- The length of each packet. 
type (character) -- The type of message passing to use: 
'n' - Trollius network level, 
'v' - Trollius virtual circuit, or 
h' - Hardware.

When done, the elapsed time, in seconds, that the OTB process waited, is printed. The prompt then appears to begin the next experiment. */

main()
{
    /* Carries the control information described above. 
       The header control.h has the structure. */
    struct control control;
/* Will store initial and final elapsed times. */
struct timeval tv0, tv1;

char c;

/* An OTB process must call kinit() to use Trollius services. */
if( kinit(0) )
{
    terror("Host doing kinit(0)");
    exit(-1);
}

while(1)
{
    printf("Enter HEX --- count length [nvh]: ");
    scanf("%x%x%Is", &control.count, &control.length, &c);

    control.type = ( c == 'n' ? 0 :
                    c == 'v' ? KHOLD :
                    c == 'h' ? THOLD : -1 );

    if( control.type == -1 ) continue; /* Error on input. */

    /* Record initial times. */
    gettimeofday(&tv0,NULL);

    /* Send the root node the control information. */
    ch_send(root,&control,sizeof(control),DINTMSG);

    printf("waiting...");
    fflush(stdout);

    /* Receive notification that the message passing is completed. */
    ch_recv(root,&c,1,0);

    /* Record final times. */
    gettimeofday(&tv1,NULL);

    /* Display OTB time calculation. */
    printf("  Elapsed time (sec)= \n",
            ((tv1.tv_sec - tv0.tv_sec) * 1000000 +
             tv1.tv_usec - tv0.tv_usec )/1.0e6 );
}
D.3 Root Code

/* Source file  root.c  for the sample TMAP program. */

#include "root.h"  /* The vnode header file. */
#include "control.h"  /* Specific to this example. */

main()
{
    /* Carries the control information described in Host.c . */
    struct control control;

    /* This is the type of message that is used.
       Initially, it is the Trollius network level. */
    int type = 0;

    /* To act both as a message and the buffer to receive a message. */
    char *buf;

    while(1)
    {
        /* Get the control information from the host. */
        ch_recv(host,&control,sizeof(control),0);

        /* Pass the control information to the next in the ring. */
        ch_send(next,&control,sizeof(control),type);

        /* Receive the same information back from the last in the ring!
           It is easier to do this, than to arrange for the last in the
           ring not to send it on to the root. */
        ch_recv(previous,&control,sizeof(control),type);

        /* Now that everyone has the control information, set
           the type of message level to use. */
        type = control.type;

        /* Create room for receiving. */
        buf = (char*) malloc(control.length);
while( -- control.count >= 0 )
{
    /* These 2 lines pass the message once around the ring. */
    ch_send(next,buf,control.length,type);
    ch_recv(previous,buf,control.length,type);
}

    /* Send acknowledgement to host. */
    ch_send(host,&type,1,0);

    /* Prepare for the next round of communication. */
    free(buf);

} /* while(1) */

} /* main */

D.4 Node Code

The code for the node processes differs from that for the root process in that it does not interact with the Host, and the recv-send pairs are reversed; otherwise all the nodes would block waiting for its neighbour to send!

/* Source file node.c for the sample TMAP program. */

#include "node.h"     /* The vnode header file. */
#include "control.h"  /* Specific to this example. */

main()
{
    /* Carries the control information described in Host.c. */
    struct control control;

    /* This is the type of message that is used.
        Initially, it is the Trollius network level. */
    int type = 0;

Appendix D. Sample Source Files

/* To act both as a message and the buffer to receive a message. */
char *buf;

while(1)
{
    /* Receive and pass on the control information. */
    ch_recv(previous,&control,sizeof(control),type);
    ch_send(next,&control,sizeof(control),type);

    /* Now that everyone has the control information, set
    the type of message level to use. */
    type = control.type;

    /* Create room for receiving. */
    buf = (char*) malloc(control.length);

    while( -- control.count >= 0 )
    {
        /* These 2 lines pass the message once around the ring. */
        ch_recv(previous,buf,control.length,type);
        ch_send(next,buf,control.length,type);
    }

    /* Prepare for the next round of communication. */
    free(buf);
}

}/* while(1) */

}/* main */

D.5 Substitutions

A substitution file is typically created by modifying the current file being used/output by TMAP.
Appendix D. Sample Source Files

MapInit.l

This first file, MapInit.l, was created by modifying MapInit, which consisted only of the first line, which maps the OTB processes to the host. The remaining 4 lines were added to tweak the embedding not to have any dilations.

10 -> -1
0 -> 2
9 -> 3
1 -> 25
8 -> 40

LoadLst.f

In the original file, LoadLst, each string is empty. An 'f' was added for vnode 10, so that that host process would be loaded in the foreground.

: 0 ""
: 1 ""
: 2 ""
: 3 ""
: 4 ""
: 5 ""
: 6 ""
: 7 ""
: 8 ""
: 9 ""
: 10 "f"
Publications:


Awards:

NSERC Post Graduate Scholarship, 1976-79. Held at Cornell University.

NSERC Post Doctoral Fellowship, 1979-81 and 1983-88. Held at UBC.