Graphical Programming for the Transputer*

Ben Abbott, Csaba Biegl, Richard Souder, Ted Bapty, and Janos Sztipanovits
Department of Electrical Engineering
Vanderbilt University
Nashville, TN 37235

Abstract

The Multigraph programming environment provides an very high level programmers interface for development of parallel and real-time processing systems. It is specifically targeted for large systems wishing to integrate a knowledge-based synthesis technique with standard numerical techniques. The result is a graphical editing environment where the user models the structure of his desired computation. Subsequently, symbolic techniques are used to translate this model to a large-grain dataflow graph. This paper describes the concepts and use of the Multigraph programming environment on a tightly coupled parallel processing platform, the IN-MOS Transputer.

Introduction

Development of real-time instrumentation systems requires the crossover of two distinct engineering fields: signal processing and computer engineering. The high level problem is that of signal processing issues, yet the computational complexity of the real-time events and parallel processing synchronization causes numerous low level, computer engineering, issues to emerge.

We have chosen to combine numeric and symbolic processing to ease the crossover of the two disciplines [1]. Symbolic techniques are used to model the interaction of the signal processing system components. The components are implemented in standard numeric languages, Fortran, Ada, C, etc. Then, a mapping function transforms the symbolic representation of the processing system to a dynamically scheduled large-grain dataflow graph, the nodes of which are the numeric components while the structure is described symbolically.

For the declarative representation of a signal processing system, mapping the symbolic structure of a system to a large-grain dataflow graph proves to be merely a depth first tree traversal. Using this mapping technique, several very high level development environments have been produced [2], [3], [4]. These environments combine the use of symbolic programming and other techniques developed in the artificial intelligence field with classic numerical approaches for solving instrumentation problems. The result is a graphical editing environment where the signal flow graph is developed. This environment is set up to allow hierarchical design of the graph components. That is, an item such as a spectrum analyzer may be built up from some previously developed lower level items (bandpass filters, square-law devices, low pass filters, etc.). This spectrum analyzer may then in turn be used to build up even higher level components. An example is shown in Figure 1. The lowest level components of the hierarchy (primitives) are implemented in standard numerical languages. When design of the top level signal flow graph of the hierarchy is completed, a mapping process converts it first to the lowest level components of the hierarchy and then maps the result into a parallel execution environment provided by the Multigraph kernel. The resulting execution ready program is inherently parallel as a result of the declared signal flow graph being parallel. As well, it has been automatically generated from the high level declarations of the signal processing system. Thus, the computer engineering issues around synchronization have been removed.

The implementation of this environment is a layered architecture at the bottom of which in HARDWARE is a heterogeneous platform of computing elements. Above this rides a SYSTEM LAYER providing multitasking, synchronization, and communication among various components. The MODULE LAYER lies above the SYSTEM LAYER and provides a virtual machine for the basic numerical calculations and the interactions and synchronization of these components. This is done by providing functions to dynami-
The KNOWLEDGE-BASED LAYER (sometimes called the system builder) is the top layer and allows the user to define modeling information such as declarative and graphical descriptions for the system. This declaration is then transformed into a data-flow graph in which each of the nodes is a numerical computing element. The connections of the graph itself establish the scheduling requirements for this parallel algorithm. The system builder then maps the built up graph down to the MODULE LAYER where it may be executed under the graph scheduling facilities provided by the kernel. It is important to note that the MODULE LAYER hides the lower levels and thus porting to another hardware platform requires only development of the SYSTEM and MODULE LAYERS. This environment has been previously implemented on multiple VAX systems, SUN workstations, IBM 9000, and HP workstations.

The INMOS Transputer

The processing platform used for the bulk of the work described in this paper consists of five INMOS T800 Transputers coupled with an Intel 386 based PC. The INMOS Transputer family [5] provides a set of microprocessors designed specifically for parallel processing systems. As such, each Transputer is a form of building block that may be easily connected to other Transputers. The 10 MIP, 1.5 Mflop, T800 is the most common member of the Transputer family. It is a 32 bit processor along with an internal 64 bit floating point unit. Due to the parallel processing target, Transputers have several unique features:

1. The Transputer hardware supports a process model with a time sliced scheduler. Context switching is in the sub-microsecond range.

2. A point-to-point synchronized intra-processor communication mechanism is supported. It provides a single machine instruction to cause a process to hang, waiting for communication until another process wants to communicate with it. After the rendezvous has been made, the communication is accomplished by a block copy from the sender's process space to the receiver's process space. Once the data transfer is complete, both processes are allowed to continue in a time-sliced fashion. The specific memory location used to tag the desired communication is called a channel. For flexibility, a process may check a set of channels to see if it will hang up waiting for communication. As well, timeout facilities are supported by the hardware.

3. Channels (described in the previous item) are also supported for inter-processor communication. In this case they are associated with an instruction overlapping DMA controller that converts a block of continuous memory bytes to a synchronized high speed serial transfer and vice-versa for the receiver. These special channels for inter-processor communication are called links. T800 Transputers support four output links and four input links. These links are capable of simultaneously transferring continuous data in both directions at 20 mega-bits per second.

4. A priority mechanism supports two levels of processes, high and low. High priority processes are not time sliced. They work as coroutines and are subject to task switches only as a result of particular instructions that wait on events such as communication. Low priority processes are time sliced on one millisecond intervals and are serviced in a round-robin fashion.
rectional point-to-point communication links, multiplexing and hopping is required for messages to go from any processor to another one. To face this issue an underlying communication system much like that of [6] was implemented. This communication system was set up to allow any two processors (including the 386) to communicate.

- The interface provided for high level user tools requires a version of the kernel with a Common Lisp interface. However, due to memory requirements, Lisp does not map well to the Transputer. The solution to this problem fell out nicely from the choice of mapping. That is, since a distributed set of subgraphs was to be supported, supporting one more subgraph on the host processor was all that was necessary. Thus, the 386 version of the kernel supports the Lisp interface necessary for the high level tools.

- There is a need to be able to dynamically link a new subprogram to the currently executing program on any single Transputer. To face this, modifications to the standard linker were made and a special runtime loader was developed.

The Transputer Multigraph

As was noted in the introduction, moving the Multigraph to a new hardware platform concerns only the SYSTEM and MODULE layers. Specific issues and the approach implemented for the Transputer version of the Multigraph kernel include:

- There are two approaches that may be used to implement the graph scheduling functions across multiprocessors: (1) The processors may be treated as a set of "anonymous workers". In this case, a central controller schedules the whole graph by simply "farming" out work (nodes of the graph to be fired) and receiving results back from each of these processors. (2) Each processor is responsible for scheduling a subgraph of the overall graph. Those arcs of the graph that are split across processors result in inter-processor communication.

Since Transputers have only distributed memory, the second option was preferred. Figure 2 shows an example of a graph split across processors in this fashion.

- A communication system is required in order to provide a virtual machine across multiple Transputers. Since Transputers provide only four bidi-

The Transputer Multigraph has been tested with various applications including: a neural net simulator, beamforming, on-line signal analysis and monitoring, and an image processing system. The third author was responsible for testing various image processing scenarios on the Transputer. His general approach was to develop the code on a mono-processor workstation running the Multigraph environment and then to test the performance on the Transputer version.

Two particular portions included: a two-dimensional convolution (filtering), and an adaptive histogram equalization technique. They each used a form of farming for load balancing in that the computational graph began by splitting the image into sub-images where they could then be parcelled out to worker actors for the specific computational task. When finished, the processed sub-images were pasted back together. The farming technique providing convolution allowed parameters to specify the size and number of pieces into which the image should be broken, while the adaptive histogram farmer was fixed so as to split the image into four sub-images only.

Demonstration System

The Transputer version of the Multigraph has been tested with various applications including: a neural net simulator, beamforming, on-line signal analysis and monitoring, and an image processing system. The third author was responsible for testing various image processing scenarios on the Transputer. His general approach was to develop the code on a mono-processor workstation running the Multigraph environment and then to test the performance on the Transputer version.

Two particular portions included: a two-dimensional convolution (filtering), and an adaptive histogram equalization technique. They each used a form of farming for load balancing in that the computational graph began by splitting the image into sub-images where they could then be parcelled out to worker actors for the specific computational task. When finished, the processed sub-images were pasted back together. The farming technique providing convolution allowed parameters to specify the size and number of pieces into which the image should be broken, while the adaptive histogram farmer was fixed so as to split the image into four sub-images only.
Figures 3 and 4 are screen dumps from the graphical editing environment used to design the convolution test. The first of these shows the top level model of the signal flow graph. Upon start-up input, it reads the filter coefficients and the image to be filtered, passes these to the filter which accomplishes the convolution and passes the results on to be rescaled and stored. The icons: read-filter, read-image, rescale, and write are primitives icons. That is, each of them represents a subroutine written in a standard programming language (C in this case). The icon filter is an abstraction of the signal flow graph modeled in figure 4, Transputer-filter. All of the computation icons of Transputer-filter are primitives. The icons Env386, EnvTl, Tsk386, ... are used to associate a particular processor with a particular computation task.

**TABLE 1**
FILTERING BENCHMARKS
(128 x 128 image - 9 x 9 filter - 4 subimages)

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>80386/20 MHz</td>
<td>56.19 seconds</td>
</tr>
<tr>
<td>1 - T800/20 MHz</td>
<td>21.44 seconds</td>
</tr>
<tr>
<td>4 - T800/20 MHz</td>
<td>5.75 seconds</td>
</tr>
</tbody>
</table>

\[ speedup = 21.44/5.75 = 3.72 \]

**TABLE 2**
ADAPTIVE HISTOGRAM EQUALIZATION BENCHMARKS
(128 x 128 image - 30 x 30 block - 4 subimages)

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>80386/20 MHz</td>
<td>598.29 seconds</td>
</tr>
<tr>
<td>1 - T800/20 MHz</td>
<td>86.92 seconds</td>
</tr>
<tr>
<td>4 - T800/20 MHz</td>
<td>28.03 seconds</td>
</tr>
</tbody>
</table>

\[ speedup = 86.92/28.03 = 3.10 \]

**TABLE 3**
ADAPTIVE HISTOGRAM EQUALIZATION BENCHMARKS
(200 x 200 image - 3 x 3 block - 4 subimages)

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>80386/20 MHz</td>
<td>1967.70 seconds</td>
</tr>
<tr>
<td>1 - T800/20 MHz</td>
<td>329.71 seconds</td>
</tr>
<tr>
<td>4 - T800/20 MHz</td>
<td>83.56 seconds</td>
</tr>
</tbody>
</table>

\[ speedup = 329.71/83.56 = 3.95 \]
Benchmarks are provided in Tables 1 through 3. Another copy of these tables as well as particulars about the specific image processing algorithms may be found in [7]. The best performance scenario yielded a speedup of 3.95 out of a theoretical 4.0 thus indicating that the processing overhead of the Transputer kernel is negligible in this application. Also impressive was the speedup of about 5 that the one transputer had over the 20 Meg. Hz 386/387 machine alone.

Conclusion

The most important result gained from this research is the outstanding performance achieved by the Multigraph architecture when mapping to a tightly coupled execution environment. This certainly states that knowledge-based system integration techniques provide a powerful tool for development of parallel systems.

The Multigraph tools remove an incredible portion of the software engineering burden associated with parallel processing, as well as large system development. It is interesting to note that the developer of the image processing demonstration system described in the previous chapter had no prior experience in parallel processing.

References


