PREP-P: A MAPPING PREPROCESSOR FOR CHiP COMPUTERS

Francine Berman
Department of Electrical Engineering and Computer Sciences
University of California, San Diego
La Jolla, California 92093

Michael Goodrich
Charles Koelbel
W. J. Robison III
Karen Showell
Department of Computer Science
Purdue University
Lafayette, Indiana 47907

Abstract

The mapping problem arises when the communication graph of a parallel algorithm exceeds in size or differs in structure from the interconnection architecture of the intended parallel machine. In this paper, we describe the design and implementation of a preprocessor which "solves" the mapping problem for CHiP architectures. In particular, we describe the mapping and multiplexing protocols of this software, and briefly discuss some preliminary test results.

0. Introduction

A fundamental activity in computer science is the implementation of algorithms on machines. This is a straightforward task for the user in the sequential environment where the algorithm can generally be specified in a high level language without regard to the size of the intended machine (available memory and peripheral storage) or its architecture. In the parallel environment, the user must specify the algorithm in an architecture-dependent manner which takes into account the number of processors in the machine and its communication architecture. More specifically, the *mapping problem* in parallel computation arises when the parallel algorithm to be implemented requires more processes than the target parallel machine has available, or requires a different communication structure than the hardwired interconnection architecture of the machine, or both.

In [3], a general strategy for mapping large-sized parallel algorithms into fixed-size parallel architectures was proposed. The goal was to find a uniform, non-ad hoc method which solved the mapping problem for the regular communication structures and non-shared memory architectures most frequently used in parallel computation. The basic idea of the general method was to first *contract* a large-sized parallel algorithm G into an intermediate graph G' which has the same communication structure but fewer processes, and then to *lay out* the intermediate graph (G') on the target interconnection structure H (Figure 1). The (large-sized) parallel algorithm is then *multiplexed* on the interconnection architecture of the target machine.

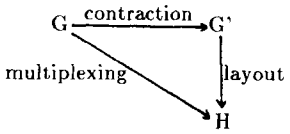

Figure 1
Mapping G into H using the general method.

Asymptotic analysis showed this method to produce optimal or near-optimal mappings for a varied group of benchmark examples. In almost every example, the automatic mapping produced by the general method distributed the algorithm's processes and communication paths roughly equally over the target architecture, with full speed-up of the contracted parallel system. These results were very encouraging but asymptotic, and we were interested in gauging the actual performance of a mapping preprocessor based on these ideas.

To this end, we designed and implemented Prep-P as a mapping preprocessor to implement the method in [3]. We chose as our target machine the family of CHiP computers ([8]), a family of non-shared memory architectures which can be configured to emulate many commonly used parallel interconnection structures. Prep-P runs as a preprocessor to both Poker [9], a software emulator of a 64 processor (called PEs) CHiP machine, and to Pringle, the CHiP hardware prototype [6].

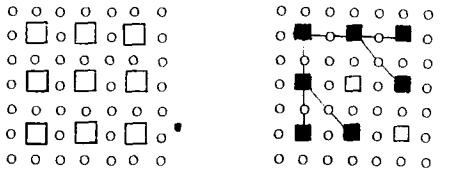

Figure 2
a) A CHiP machine.
b) A CHiP machine configured as a tree machine.
(Boxes represent PEs (processors); circles represent switches).

1. Design

The goal of Prep-P is to allow a user to input a parallel algorithm to a CHiP machine regardless of size and structure. Prep-P then acts as a front-end interface between the user and the CHiP machine, creating a mapping and performing the multiplexing. The output of Prep-P is 8051 assembly code executable on Poker or Pringle (the CHiP architecture emulators) which simulates the execution of the original parallel algorithm.

The design of Prep-P follows the general strategy described in [3], i.e. the program is divided into *user interface, contract, layout* and *multiplexing* modules. We briefly describe the design of each of the modules of Prep-P.

A. User Interface (Input)

Input to Prep-P consists of a parallel algorithm represented by an undirected communication graph with integer nodes. Each node is assosciated with a not necessarily distinct process written in XX, the CHiP parallel programming language [9]. The communication graph can be input as an adjacency list. however since many parallel interconnection structures are regular and modular, the communication graph may also be given by a list of arithmetic expressions specifying for each node a set of incident nodes (and their assosciated processes).

For example, if the communication graph of a broadcast algorithm is a complete binary tree, then we might use processes *LEAF, INTERNAL* and *ROOT* to represent the functions of the leaves, non-root internal nodes and root node respectively. Assume a straightforward numbering scheme for the tree, i.e., let the root be 1 and for node i, number its leftson 2i and its rightson 2i+1 (Figure 3). An input specification to Prep-P for the max-finding algorithm is given in Figure 4.

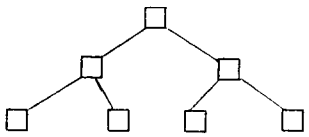

Figure 3
Interconnection structure for the broadcast algorithm.

```
tree
nodemin = 1
nodecount = 7
procedure ROOT
        nodetype: {i == 1}
        port RSON: {2*i}
        port LSON: {2*i+1}
procedure INTERNAL
        nodetype: {i> 1 && i≤ 7/2}
        port FATHER: {i/2}
        port LSON: {2*i}
        port RSON: {2*i+1}
procedure LEAF
        nodetype: {i> 7/2}
        port FATHER: {i/2}
```

Figure 4
Prep-P specification for a broadcast algorithm.

The first three lines in the code specification in Figure 4 indicate to Prep-P the graph type and node range. Distinct adjacencies of a vertex are distinguished by assigning each incident edge a distinct port label.

Given a parallel algorithm input in this format, Prep-P creates internal files which represent the adjacency and process information in a more convenient form. If the number of nodes in the communication graph of the algorithm is greater than 64, control is next passed to the contraction module, otherwise control is passed to the layout module.

B. The Contraction Module

Currently, the contraction module takes as input an undirected graph and outputs a smaller-sized (contracted) intermediate graph of the same graph type. The class C of acceptable input communication graphs includes cube-connected cycles, toruses, hex and square meshes, linear arrays, complete binary trees, loops, shuffle-exchanges (which contracts to a 4-pin shuffle), butterfly networks, 4-pin shuffles, finite element graphs, and hypercubes.

The contraction module consists of a set of library routines which are based on the mappings produced by edge grammar representations of graph families in C. (Edge grammars ([1], [2]) are formal systems similar to graph grammars which can be used to define and automatically contract graph families). During the contraction phase, a number of internal files are generated including files which list the adjacency structure of the contracted graph and give the mapping of nodes in the algorithm's communication graph to nodes in the contracted intermediate graph. Recall that the contraction module is only entered if the communication graph of the input algorithm has more than 64 nodes.

We are now designing an improved contraction procedure which will enlarge the class of acceptable input graphs to include any undirected graph. Using this new procedure, we intend to combine both layout and contraction modules to one mapping module.

C. The Layout Module

After contraction, or if the input graph has less than 64 nodes, control is passed to the layout module. The layout module embeds the contracted communication graph into a 64 PE grid and then uses a shortest path algorithm to route the edges through the switch lattice. The placement algorithm is based on a divide-and-conquer strategy which iteratively embeds partitions of the contracted graph into quadrants of the the PE lattice. The partitions are chosen so as to minimize the maximum distance (wirelength) of the nodes in distinct quadrants. More specifically, the placement algorithm proceeds as follows:

1) Initially partition the contracted graph into subgraphs G_1, G_2, G_3 and G_4 and the PE grid into quadrants Q_1, Q_2, Q_3 and Q_4.
2) Reformulate the $\{G_i\}$ so that the number of edges between vertices in distinct subgraphs is minimal. Call these reformulated subgraphs G_1', G_2', G_3' and G_4'. (If the contracted graph has a separator [7], this is straight-forward. For a more general class of graphs, the problem is NP-complete [4] and a heuristic must be used. Our algorithm uses a Kernighan & Lin-type heuristic [5]: Essentially switch pairs of vertices in distinct partitions (quadrants) until the number of edges between vertices achieves a local minimum).
3) Find a mapping of the subgraphs G_j' into the quadrants Q_i that will minimize the maximum wirelength between subgraphs. If more than one of the 24 possible mappings achieves a minimum, then take the one which also minimizes the total wirelength between subgraphs. For each i, let Q_i' be the quadrant in which G_i' finally resides.
4) Apply the algorithm iteratively to each partition/quadrant pair $< G_i', Q_i' >$ until $|Q_i'| = 1$. When $|Q_i'| = 1$, use the trivial embedding.

When the nodes of the contracted graph have been placed on a 64 PE grid using the placement algorithm, the edges are then routed on the switch lattice of the CHiP using a shortest path algorithm. If the nodes in the contracted graph have degree greater than 8 (maximum PE degree for the CHiP architecture), the layout algorithm can be modified to couple nodes in the PE lattice.

After layout and contraction, the communication graph has been embedded in the target architecture. Internal files have been generated which will communicate the CHiP layout to Poker or Pringle and which will define for each PE the set of processes in the original graph to be simulated at that location. It remains to multiplex the process codes using this mapping.

D. The Multiplexing Module

In CHiP programs, communication is accomplished by simple reads and writes. During contraction and layout, distinct processes which communicate are either mapped to the same PE *(intra-* processor communication) or to different PEs *(inter-* processor communication). To simulate the original communication without conflict, Prep-P substitutes read or write macros for each simple read or write instruction. The multiplexing is then performed by concatenating the codes of the input processes assigned to each PE in the CHiP lattice, and by executing the concatenated process codes for each PE in a round-robin fashion (context-switching at each I/O call). Communication of the input algorithm is simulated by maintaining local mini-operating systems at each PE which update a set of arrays and buffers needed for I/O management. These protocols are straightforward but seem to be robust.

3. Implementation and Preliminary Results

Prep-P is written in a combination of shellscript, C, XX and Intel 8051 assembly language. The first version was brought up on a Vax 780 using a BBN Bitgraph graphics terminal. To coordinate with the newest distribution version of Poker, the current version was brought up on a Sun 2.

We are currently testing and improving Prep-P. Among the preliminary benchmarks, we give timing results for three parallel algorithms with distinct communication paradigms and interconnection architectures. They are the FFT (on a butterfly network), a broadcast algorithm (on a complete binary tree), and a pipelined algorithm (on a linear systolic array). Figure 5 shows a table of preliminary timing results.

At current status, we are continuing to test and improve Prep-P with the dual goals of distributing the software for experimental use, and using this experience to create efficient mapping protocols to be used with other types of parallel architectures.

Algorithm	un-multiplexed G(n)	multiplexed G(n+1)	multiplexed G(n+2)
Broadcast	8407 ticks (63 nodes)	32808 ticks (127 nodes)	45448 ticks (255 nodes)
Pipelined	35301 ticks (64 nodes)	177177 ticks (128 nodes)	318857 ticks (256 nodes)
FFT	5061 ticks (32 nodes)	31109 ticks (80 nodes)	*

Figure 5
Timing results for large-sized algorithms reduced by Prep-P and executed by Poker. (1 tick = 1 microsecond on Pringle).

Acknowledgements

We are grateful to many people for their help on this project. We would like to thank Mark Anderson, Dennis Gannon, Steve Holmes, Kevin Smallwood, Larry Snyder, Joel Strickland, John Rice and Ko-Yang Wang for their support and encouragement.

Bibliography

[1] F. Berman, "Edge Grammars and Parallel Computation," Proceedings of the 1983 Allerton Conference.

[2] F. Berman and G. Shannon, "Edge Grammars: Decidability Results and Formal Language Issues," Proceedings of the 1984 Allerton Conference.

[3] F. Berman and L. Snyder, "On Mapping Parallel Algorithms into Parallel Architectures," (extended abstract) Proceedings of the 1984 International Conference on Parallel Processing.

[4] M. Garey and D. Johnson, *Computers and Intractability: A Guide to NP-Completenes*, W.H. Freeman and Co., 1979.

[5] B. Kernighan and S. Lin, "An Effective Heuristic Procedure for Partitioning Graphs," *Bell Systems Technical Journal*, 49(2). February, 1970.

[6] A. Kapuan, K.Y. Yang, D. Gannon, J. Cuny and L. Snyder, "The Pringle: An Experimental System for Parallel Algorithm and Software Testing," Proceedings of the 1984 International Conference on Parallel Processing.

[7] R. Lipton and R. Tarjan, "A Separator Theorem for Planar Graphs," *SIAM Journal of Applied Mathematics*, vol. 36, #2, April, 1979.

[8] L. Snyder, "Introduction to the Configurable, Highly Parallel Computer," *Computer*, January, 1982.

[9] L. Snyder, "Introduction to the Poker Parallel Programming Environment." Proceedings of the 1983 International Conference on Parallel Processing.

* To multiplex an FFT of size 192 (G(n+2)) on the CHiP, a different contraction must be given than the one currently in the contract module's library routines. Alternatively, the buffer arrays could be extended to simulate 16 processes at one PE.