AUTOMATIC GENERATION OF PARALLEL PROGRAMS FOR MIMD COMPUTER SYSTEMS

V. A. Kostenko

INTRODUCTION

Four realistic approaches to development of application software for multiprocessor computer systems (MCS) have been identified in [1]: extension of existing languages; extension of compilers to include tools for generation of parallel programs; addition of a new language level above an existing language; definition of an absolutely new language with a new compiling system. The authors of [1] particularly recommend the second method, because it relieves the programmer from the responsibility for synchronization, relieves the user from the need to adapt applications to specific tasks, ensures independence of MCS architecture. Yet it requires a very complex compiler capable of automatic generation of parallel programs.

Automatic generation of parallel programs for MCS with MIMD (many instructions, many data streams) architecture in general requires solving two problems:

1) elucidation of parallelism allowed by the original program;
2) generation of parallel branches (processes).

The first problem is solved independently of the architectural features of the MCS, and the result of its solution is some representation of the parallelism that the application admits. This representation can be unified for various MIMD MCS by adopting a standard for representation of allowed parallelism. The need to solve the first problem is eliminated if the original program is defined in a parallel language or if the task is to adapt an existing parallel program developed for some MCS to work in the architecture of another MCS.

The solution of the second problem always requires taking into consideration the architectural features of the MCS for which the parallel program is being developed. It may be formulated in the following terms:

a) as generation of a parallel program with minimum execution time, which requires straightening the critical path into a single branch;

b) as generation of a parallel program with execution time not exceeding a specified limit and minimum use of resources, which requires hardware load optimization.

Most available methods for identification of parallel branches assume that the original program is defined by a directed acyclic graph $G = (V, E)$ of the application algorithm [2]. This graph is obtained by solving the first problem. The vertices $V = (v_i, i = 1, M)$ of the graph $G$ correspond to the execution of statements (macroconstructs) of the original program. These are the parallelization units, each with a known execution time $t_{i}^{op}$ or a known expected execution time $t_{i}^{op}$. The arcs $E = (e_{ij}, i, j = 1, M)$ of the graph represent the information dependences between the parallelization units. Given this representation of the parallelism admitted by the original program, the solution of the first problem involves transformation of the logical structure of the original program, which contains loops and branchings. The logical structure is "embedded" in separate macroconstructs or "unrolled" [3-5]. The original program is thus reduced to a linear program without loops and branchings.

The feasibility of this representation of parallelism for a wide class of applied programs has been considered in [6].

All methods of identification of parallel branches can be divided into two large groups.

1. Methods based on mapping (embedding) the algorithm graph on the MCS graph [2, 5]. This requires solving the following problems: establishing a homomorphism between two graphs, homomorphic folding of a graph into a graph with specified properties, decomposition of the vertex set of a graph into subsets forming separate paths, etc. All these problems

are NP-complex and the time to solve them often is not less than the time for sequential execution of the original program. These methods are therefore applied only for MCS architecture design or for analysis of potential parallelism admitted by various classes of applications.

2. Methods based on elucidation of objective functions (criteria) that characterize the property of the parallel program and their optimization in the course of allocating the original program statements to branches [3, 4, 7-13], as well as critical path methods [14]. The low operating complexity of these methods compared with the methods of the first group makes them promising for use in compilers. The methods differ from one another by the type and complexity of the objective function, universality (classes of MCS to which the method is applicable and original program constructs that may be used as parallelization units), the strategy of traversal of the graph vertices, and the quality of the resulting programs. This group of methods includes the algorithms for identification of parallel branches proposed in Sec. 4.

1. DESCRIPTION OF MCS ARCHITECTURE AND PARALLEL PROGRAMS

A sufficiently full description of the MCS architecture for purposes of automatic generation of parallel programs is provided by the connectivity matrix of the nodal processors (NP) among themselves $S_{ij} = [s_{ij}, i, j = 1, N]$, the NP instruction set $O_j = \{o_i, i = 1, no_j \}$ with corresponding execution times $\tau_j = \{\tau_i, i = 1, no_j \}$, and the MCS memory allocation matrix [3]. Here $N$ is the number of NPs in the MCS, $N_{NP}$ is the number of NPs of different types in the MCS, $no_j$ is the number of instructions accepted by NP of type $j$.

If the MCS consists of $N$ NPs, then the original sequential program is partitioned into $N$ branches, which are executed concurrently with appropriate synchronization and exchange of required data. Each branch is assigned to its own NP, on which it is executed sequentially. The reduced length of branch $j$ is the total time $t_j$ to execute all the statements of the particular branch on one NP:

$$t_j = \sum_{i=1}^{M_j} t_{ij}^{op} + T_j^{ex} + T_j^w,$$  \hspace{1cm} (1)

where $M_j$ is the number of statements in branch $j$, $t_{ij}^{op}$ is the execution time of statement $j$ in NP, which can be defined in terms of the elements of the set $T$, $T_j^{ex}$ is the time spent exchanging data with other branches, $T_j^w$ is the idle time of branch $j$ while waiting for data and conflict resolution due to the use of shared resources. A parallel algorithm terminates when all its branches terminate, and the execution time $T_{MCS}^w$ of a parallel algorithm is thus given by

$$T_{MCS}^w = \max_{j=1, N} t_j. \hspace{1cm} (2)$$

The height of statement $i$ in branch $j$ is defined as the initialization time of this statement:

$$t_{ij} = \max \left\{ \max_{l=1}^{ob} t_{il} + \sum_{l=1}^{i-1} t_{lj}^{op} + T_j^{ex}, <i + T_j^w, <i \right\}, \hspace{1cm} (3)$$

where $t_{ij}^{ob}$ is the time for statement $i$ to obtain all its operands (more precisely, the time to obtain the last operand); the second value in max(·) is the execution time of all the statements preceding $i$ in the branch.

2. QUALITY CRITERIA FOR PARALLEL PROGRAMS

An ideal parallel program satisfies the following conditions:

- all the branches are of the same length $t_1 = t_2 = \ldots = t_N$ with optimal hardware loading, and the lengths of all branches do not exceed the length of the critical path for minimum execution time;
- there is no idle time due to waiting for data and conflict resolution associated with the use of shared resources ($T_n^w = 0, n = 1, N$);
- data exchange between branches overlaps computation ($T_n^{ex} = 0, n = 1, N$).