The emergence of inexpensive hardware platforms has lead to an increasing interest in concurrent processing systems, where the processing is not limited to a single CPU. Such systems offer the promise of faster execution times, increased system reliability, availability and fault-tolerance, as well as the ability to solve hitherto unsolvable problems - those that require an immense amount of computational power. However, while conventional uniprocessor systems require a sequential program representation as input, these systems require a completely different approach to problem formulation - one that fully exploits the concurrency aspects of the problem. In this paper, we present the issues in concurrent program execution, and survey some of the current research in this area.

Background

Parallel processing tries to exploit the concurrency of different events within a system during the computational process. This can be done at three different levels:

- at the job or program level,
- at the inter-instruction level, and,
- at the intra-instruction level.

The highest level is usually implemented through multiprogramming, timesharing and multiprocessor systems. Multiprogramming and timesharing exploit concurrency within a uniprocessor by simulating a multiprocessor system. Usually, the parallelism at this level is provided by system software, with the programs (jobs) themselves consisting of sequential algorithms implemented in a sequential language, i.e., one with no parallel constructs. At the inter-instruction level, parallelism is introduced by means of parallel algorithms and parallel languages, or more usually by adding a few parallel constructs to a strictly sequential language. Parallelism at the intra-instruction level involves finding the potential parallelism within a single instruction, and is usually implemented through hardware means. As we move from the topmost level to the bottommost, an increasing amount of the functionality is implemented in hardware, and the “grain-size” of parallelism decreases.

The Hierarchical Program Model

A program consists of an algorithm which specifies the nature and sequence of the various sub-tasks to be performed. Each of these tasks is composed of one or more high-level language statements, and each of these statements consists of one or more machine language instructions. Each machine language instruction causes state transitions within the underlying hardware [3]. Thus a single program has different representations at different levels, although all of the representations use the same resources. Since parallel processing ultimately attempts to optimally use multiple hardware resources concurrently, our goal should be to obtain the most efficient and appropriate machine level representation of the program. So the representations at other levels should be geared towards generating a machine language representation that allows the maximal amount of parallelism possible. This implies that we should not aim to find the most efficient representations at higher levels, since they may constrain the extent of parallelism possible at the lowest level. Rather, the goal is to find higher level representations that lead to an optimally parallel set of machine and hardware state transitions.

However, there exists a very large base of widely available sequential programs (sequential algorithms, implemented in a sequential language) so that a lot of effort has been expended on developing parallelizing compilers. These compilers accept a sequential program as input, and attempt to find all the “task” and “loop” parallelism inherent within the program. This is usually done by analyzing the flow of control and data within the program.

Issues in Parallel Processing

There are three major issues facing parallel programming/processing:

- Representing parallelism within an algorithm or in a source language program,
- Detecting instances of parallelism in sequential programs, and
- Partitioning and scheduling the resulting parallel program on a target machine with multiple processors.

The representation problem can be handled by developing new program representation languages (source/high level languages) which have constructs that support concurrency and parallelism, or by adding such constructs to already existing procedural languages. There are three ways in which parallelism can occur in a representation:

- Undetectable, i.e., parallelism cannot be detected, either due to data dependencies or on account of a poor representation.
- Detectable, from the program’s control flow and data dependency graphs.
- Explicit, i.e., the representation contains parallelism constructs.

The detection of parallelism is usually done at the source language level. Parallelizing compilers for procedural
languages depend upon various forms of program dependency graphs to establish the control flow and data dependencies within a program. This is based on the assumption that a functional representation is more likely to expose potential parallelism than the conventional sequential program model. These compilers then try to detect 'loop' (horizontal) and "task" (vertical) parallelism from the graph. In loop parallelism, different iterations of a loop are executed in parallel, while task parallelism involves executing different loops or procedures in parallel.

Once we obtain the parallel program, either from the program representation itself or by detecting parallelism within the sequential program, we are faced with the problem of partitioning the program into sub-tasks and assigning these sub-tasks to the available processors. The 'grain-size' of partitioning (parallelism) is determined by the type of target machine architecture. "Fine-grain" parallelism is more appropriate for tightly coupled multiprocessors, where communication between processors is not too expensive but communication is many magnitudes more expensive than local computation. "Coarse-grain" parallelism is better suited to loosely coupled multiprocessors or distributed systems, where inter-processor communication is many magnitudes more expensive than local computation. Task assignment/scheduling is usually done on the basis of one or more of the following conflicting factors:

- Minimal interprocessor communications,
- Load-balancing across all available processors, and
- Minimal job execution time, or smallest response time.

Most current partitioning and scheduling algorithms make some limiting assumptions about the nature of the program, and sometimes the underlying machine architecture, and generate optimal assignments on the basis of one or more of the factors listed above.

Some Current Techniques in Parallelizing Compilers

Shen et al [2] identify three factors that weaken current parallelizing techniques:

1) Symbolic terms with non-zero values,
2) Coupled subscripts, and
3) Non-zero and non-unity coefficients of loop indices.

From their empirical study of twelve production programs written in Fortran, they conclude that:

1) User assertions and inter-procedural analysis can be used to reduce the number of symbolic terms with unknown values,
2) Most existing algorithms cannot handle multidimensional arrays, and
3) The overwhelming majority of non-zero coefficients are either 1 or -1.

Balasundaram and Kennedy [3] propose summarizing the effects of all the data dependencies between different regions in the program, and using this summary to guide the detection of task parallelism. For each region R of the program, they define a Data Window as the set of Data Access Descriptors, one for each variable accessed in the region. The Data Access Descriptor of a variable in a region contains a summary of all accesses to the variable in that region. To distinguish between accesses to different parts of an array variable, they define a Simple Section, which is bounded by Simple Boundaries. A Simple Section can represent most commonly used array access patterns such as the access of a single diagonal, or a triangular section, or even the whole array. They then use the following algorithm to detect task parallelism in a sequential program:

1) For each loop nest in the program, compute its Data Window.
2) Construct a Task Dependence Graph, G_T = (T, E) where T is the set of all loop nests in the program, and E is the set of edges connecting pairs of loops that have a dependency between them (i.e., the intersection of their Data Windows is non-empty).
3) Partition all the tasks in G_T into a sequence of sets, such that if a task "a" precedes a task "b" in topological order, the set containing "a" precedes the set containing "b" in the sequence.
4) Execute the sets in order, with all the tasks in each set being executed concurrently.

Cytron et al [4] present an algorithm for automatically generating Directed Acyclic Graph (DAG) parallelism from a sequential program which eliminates some data dependencies by introducing private variables, and enforces a partial ordering on the statements in the basis of the program's control flow graph. They use the concept of control dependency as the basis for potential parallelism, thereby allowing the detection of loop as well as task parallelism. Their algorithm is as follows:

1) Generate the control flow and data dependency graphs.
2) Use private variables for each node in the data dependency graph, thereby eliminating all edges in the graph.
3) Build a partial order of all statements in the program on the basis of the control flow graph, and eliminate those private variables that are redundant.

Cartwright and Fellesin [5] show that such a partial ordering, which they call a Program Dependence Graph (PDG), is actually a conventional data-flow graph.

Li, et al [6] have developed a new, more accurate algorithm for the data dependency analysis of multidimensional arrays, based on numerical methods using the theories of Diophantine equations and the bounds of real functions. Earlier numerical methods examined data areas accessed by two array references dimension by dimension. If the two areas representing the subscript expressions were disjoint in any dimension, there was no data dependency between the two references. However, if each pair of areas appeared to overlap in each dimension, it was not clear if there was an overlap when all dimensions were considered simultaneously. Under such circumstances, a data dependency had to be assumed, even if it did not really exist. The new algorithm considers all dimensions simultaneously, and selects a few "view angles" based on the subscripts, so that it gets an exact view of the data areas.

Polychronopoulos [7] introduces compiler optimizations for compile-time loop parallelism detection, and run-time dependency checking for loops with data-dependent parallelism. Compile-time loop parallelism detection is performed by cycle shrinking, in which cycles within the dependency graph are broken, or at least reduced, by node splitting. Run-time Dependence Checking applies to loops whose array subscripts are not amenable to compile-time analysis, or even to iterations of serialized loops that are dependence free.

497
Hendren and Nicolau [8] apply Interference Analysis to dynamic data structures. This approach approximates the relationships between accessible nodes in large aggregate data structures. These relationships are represented by path expressions, a restricted form of regular expressions. The relationships are then used in three different parallelization methods:

1) A fine-grain analysis that checks if basic statements can be executed in parallel,
2) A coarse-grain analysis to determine if procedures can be executed in parallel, and,
3) An analysis to determine if two different statement sequences interfere.

Some Current Approaches to Processor Allocation

Polychronopoulos and Banerjee [9] present two heuristic allocation algorithms, WP (Weighted Priority) and PA (Processor Allocation). WP functions as follows:

a) Compute the Critical Path (CP) of the task graph G, where the nodes represent tasks and are labelled with their execution times, by selecting the final node, and then finding a path to the initial node such that at each level, the node selected has the largest label of all the choices possible.

b) Compute task priorities as follows:
   1) Give priority to tasks that belong to the critical path,
   2) Give priority to those tasks that have the longest execution times, and
   3) Give priority to those tasks that have the largest number of immediate successors, associate weights for each of the above three rules, and sum them up.

c) Order the tasks by priority and starting from the highest priority unfinished task, select n tasks such that their processor requirements are exactly equal to or just greater than the number of available processors.

d) Use PA to allocate processors for this selected subset of tasks.

e) Repeat steps c), and d), until all tasks have been completed.

PA allocates one processor to each of the selected tasks, and then, for tasks that request more than one processor, allocates the remaining processors in proportion to the size of the task's request.

Rogers and Pingali [10] present a "data-driven" decomposition scheme for target architectures that have only two levels of memory. Their approach requires the programmer to specify a "domain decomposition" that specifies the distribution of data across the multiprocessor along with the sequential program. Their compiler then generates a process decomposition which is optimized to the nature of the program, or the nature of the underlying hardware architecture, so that they are not truly flexible.

Shin and Chen [11] try to minimize interprocess communication delays in their algorithm. They consider two graphs, the task graph $G_T$ and the processor graph $G_P$. An edge between two nodes in $G_T$ indicates the presence of direct communications between the two tasks. Now, they select processors in $G_P$ such that the "dilation" is below some acceptable value. They define dilation as the maximal number of hops between two nodes in $G_P$, to which two adjacent nodes in $G_T$ are assigned.

Polychronopoulos et al. [12] present a static processor allocation algorithm which minimizes parallel execution time for loops without branches, when the loop iterations and the number of available processors are known. They define an "efficiency index" $E_i$ for a loop with i iterations and with p processors assigned as

$$ E_i = \frac{\left\lceil \frac{i}{p} \right\rceil}{p} $$

Once they build up an efficiency table for all loops with all possible processor allocations, they compute their allocation function $G_k(P)$, which is recursively defined for a loop k as

$$ G_k(P) = \max \{ E_i^P, G_{k+1}(P) \} $$

where P is the number of processors assigned to the loop, and can vary from 1 to the total number of processors. Their algorithm then works as follows:

- Compute the efficiency table $M$ for all loops and for all possible allocations,
- Compute the allocation table, and generate the processor allocation profile.

Conclusions

Programs for multiprocessor systems require a completely different approach from those for uniprocessor systems. The effective use of multiprocessor systems requires a program representation which clearly indicates sections (tasks) that can be executed concurrently, mechanisms for synchronizing these different concurrent modules, methods for ensuring data integrity, and algorithms that schedule and allocate these modules to the available processors efficiently. Most current work in these areas make very limiting assumptions about the nature of the program, or the nature of the underlying hardware architecture, so that they are not truly flexible.

References


