Parallel Computing 16 (1990) 41–57
North-Holland

Structured partitioning of concurrent programs for execution on multiprocessors *

Chien-Min WANG and Sheng-De WANG
Department of Electrical Engineering, National Taiwan University, Taipei 10764, Taiwan

Received 8 June 1989
Revised 22 January 1990

Abstract. In this paper, the problem of partitioning of parallel programs for execution on multiprocessors is investigated. By assuming run-time scheduling approaches, the static partitioning problems are formulated and solved in the context of a structured program representation model, in which a program is assumed to be composed of parallel loops. The structured partitioning problem is then defined as the problem of partitioning each parallel loop into appropriate number of tasks such that the program execution time is minimized in some sense. The worst case of the program execution cost is adopted as the minimization criterion, so the results obtained in this paper are guaranteed to be within some performance bound. Two algorithms are developed to solve this problem. The first algorithm using a prune-and-search technique can find the optimal partition, while the second algorithm can obtain a near optimal partition for the simplified partition problem in linear time. It is proved that the cost of the partition generated by the linear time algorithm is at most twice the cost of the optimal partition.

Keywords. Program partitioning, Shared memory multiprocessor, Scheduling mechanism, Structured partition problem, Simplified structured partition problem, Cost analysis.

1. Introduction

One of the trends in developing future general-purpose supercomputers is toward the shared memory multiprocessors. However, a serious problem arises when we attempt to execute parallel programs on these multiprocessors. The overheads of communication, synchronization, and scheduling become a bottleneck to parallel processing [9,10]. Let the granularity of a parallel program be defined as the average execution time of a sequential unit of computation in the program without inter-processor synchronization or communication. For a multiprocessor system, there exists a minimum program granularity below which the performance degrades significantly due to frequent synchronizations and communications. On the other hand, the coarser granularity means the more parallelism lost and the attempt to minimize such overheads usually results in reducing the degree of program parallelism.

In order to obtain an appropriate granularity, the technique of program partitioning was proposed [2,13,18]. Program partitioning is the process of partitioning a parallel program into tasks in such a way that the inter-task communication and synchronization overheads are minimized and the resulting tasks can be scheduled onto the available processors without excessive scheduling overheads. However, finding the optimal partition is a nontrivial optimiza-

tion problem. Most instances of this optimization problem have been proved to be NP-complete [18,19].

One solution to this problem is the explicit partitioning approach which is done by the programmers, but there are several drawbacks about this approach. First, this approach places a tremendous burden on programmers - both for correctness and performance. Second, the partition defined by the programmers is unlikely to be near the optimal partition. Third, the program written with an explicit partition may not be portable in performance to difference multiprocessors. Finally, the way to express the partitioning of a parallel program is dependent on the programming language used.

In order to free programmers from the above inconveniences, the development of a software program that can automatically partition parallel programs into tasks for the target multiprocessor is desired. However, the complexity of this problem has led the computing community to adopt heuristic approaches. A major disadvantage of heuristic approaches is that they are based on some conjectures or experiences, so there is no guarantee about the worst-case performance of these algorithms. In this paper, we propose an alternative approach. Since it is difficult or impossible to find a universal solution for the general partition problem, we concentrate our attentions on two problems called the Structured Partition Problem and the Simplified Structured Partition Problem. Then two algorithms will be presented. The first algorithm uses a prune-and-search technique to reduce the time in searching the optimal partition for the Structured Partition Problem. The second algorithm will generate a near optimal partition for the Simplified Structured Partition Problem in linear time. The cost of the partition generated by the linear time algorithm is proved to be at most twice the cost of the optimal partition.

The rest of this paper is organized as follows. Section 2 gives some background information and the necessary definitions. Section 3 defines the Structures Partition Problem and the Simplified Structured Partition Problem. The two algorithms will be presented in Sections 4 and 5, respectively. Finally, conclusions are given in Section 6.

2. Background

There are three possibilities for automatic partitioning and scheduling [18]. They are the run-time partitioning and run-time scheduling approach, the compile-time partitioning and run-time scheduling approach, and the compile-time partitioning and compile-time scheduling approach. In this paper, we adopt the second approach. This approach is very attractive because it provides a good trade-off between the inaccurate estimate of the compile-time analysis and the extra overhead of the run-time analysis. In addition, we shall study the partition problem only.

As shown in Fig. 1, an automatic partitioner takes a given parallel program and a target multiprocessor model as inputs and produces a partitioned program as output. Before partitioning the parallel program, a front end will transform the parallel program into an program representation suitable for the process of partitioning. After the partitioned program is produced, a code generator is used to generate the target code. At run time, the scheduler picks ready tasks and assigns them to available processors for execution.

2.1 The multiprocessor model

The target multiprocessor is a tightly coupled multiprocessor. It is assumed that each processor has its own local memory and processors communicate through shared memory

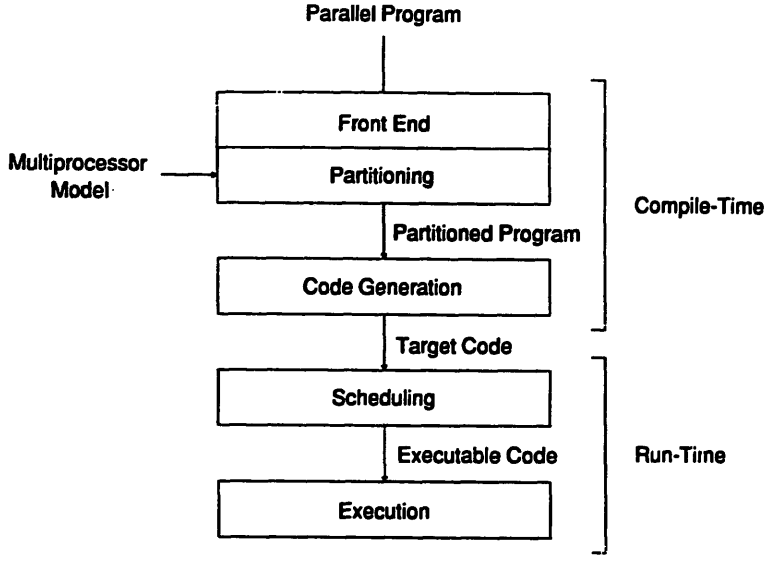

Fig. 1. The compile-time partitioning and run-time scheduling approach.

modules. Neither processors nor shared memory modules are distinguishable from their neighbors. Each shared memory module is capable of being accessed by any processor through the interconnection network. Systems in this class include the Alliant FX series, the Denelcor HEP, the University of Illinois Cedar system, the New York University Ultracomputer, and the IBM RP3 multiprocessor.

The only inter-processor interactions considered are the task scheduling and the data communication (synchronization is treated as a special case of communication). All other inter-processor interactions which may arise in the real multiprocessor systems, such as I/O contention for system resources and distribution of program code are ignored. Access to the shared memory modules is assumed to be deterministic in that either fetches or stores from a single processor are processed in the order they are issued. This means that either fetches or stores within a processor are done serially, or that an equivalent ordering is imposed on data accesses.

The performance of the target multiprocessor can be characterized by a 5-tuple $M = (P, R, W, S, E)$. Among these, P is the number of processors in the target multiprocessor. The function R is the communication overhead function for reading data. It estimates the time required to fetch data from a shared memory module to a local memory through the interconnection network. The function W is the communication overhead function for writing data. It estimates the time required to store data from a local memory to a shared memory module through the interconnection network. The function S is the scheduling overhead function. It estimates the scheduling overhead incurred by a processor when it begins execution of a new task. The function E estimates the time required to complete the computation of a program segment.

2.2 The program representation

The parallel program is represented as a program graph based on the Graphical Representation (GR) [18]. In the program graph, parallelism is explicitly specified. As discussed in the above, a front end transforms parallel programs into program graphs and computes those attributes needed for partitioning. For each transformed program, it is assumed that each branch of an IF or GOTO statement is assigned a branching probability by the user, or

automatically determined by the front end. Each loop is automatically normalized. Like the branching probabilities, unknown loop upper bounds are either defined by the user, or automatically determined by the front end.

In the program graph, a program is composed of a set of function definitions. Each function definition consists of a directed graph representing its computations. A directed graph is a 2-tuple $G=(V, A)$ where V is a set of nodes and A is a set of arcs. The nodes in V represent the computations of that function and the arcs in A represent communications and synchronizations between nodes, which enforce the precedence constraints.

There are four kinds of nodes. A simple node represents an indivisible sequential computation like a simple statement. A parallel node represents the parallel iterative construct like DOALL statements. A function node represents a function call to a function. A compound node contains a set of graph-frequency pairs and is used to denote any complex program structures. Each execution of a compound node is an arbitrary sequential execution of its subgraphs, so that subgraphs may be executed any number of time in any order. If we consider subgraphs to be like basic blocks, this general definition corresponds to a sequential flow graph which is complete and thus includes all other flow graphs (e.g. IF, WHILE). In general a program graph may contain nested DOALL loops. A program graph is a simple program graph if it does not contain nested DOALL loops.

Arcs represent the precedence constraints between nodes. In modeling precedence graphs, two language constructs are often used in parallel languages. One is the concurrent construct of the block-structured language originally proposed by Dijkstra and the other is the FORK/JOIN construct. The latter construct has the drawback that, like the GOTO statements in the sequential programs, the FORK/JOIN statements make the parallel programs hard to read and understand. Therefore, in this paper we consider the concurrent construct only. In this case, the set of n statements $S_1, S_2, \ldots, S_n$ that can be executed concurrently can be expressed as the statement PARBEGIN S_1; S_2; ...; S_n PAREND. Although the concurrent construct alone is not powerful enough to model all possible precedence graphs, it is well suited to structured programming.

Fig. 2 shows the example program used throughout this paper and its program graph. Although the program graph representation is useful for representing programs written in different languages, it is inconvenient in illustrating our partitioning algorithms. We shall use the parse tree representation instead. In a parse tree, leave nodes represent simple nodes, while internal nodes represent constructs like DOALL constructs, BEGIN/END constructs, and PARBEGIN/PAREND constructs. For a structured program graph that can be modeled by the PARBEGIN/PAREND construct and the BEGIN/END construct, it is easy to generate the parse tree of a program from its program graph [13]. The parse tree of the example program is shown in Fig. 3.

2.3 The partitioned program

The context switching, necessary for synchronization, and the task migration, necessary for load-balancing are two major sources of scheduling overhead. If they occur frequently, the scheduling overhead incurred may well undo the potential speedup obtained by parallel processing. This problem will be greatly simplified if tasks are functional in nature. A functional task can only be scheduled when all its inputs are available. Once scheduled it can run to completion without interacting with other tasks. The context switching for synchronization and the task migration for load balancing are no longer necessary during the execution period of a functional task. Scheduling overhead is only incurred when a task starts execution and can be easily predicted at compile time. Note that, for a functional task, there is only one

```
B1: PARBEGIN
    L1: DOALL I = 1,20
        L5: DOALL J = 1,4
            S1
        ENDDO
    ENDDO;
    B2: BEGIN
        L2: DOALL I = 1,48
            S2
        ENDDO;
        B3: PARBEGIN
            L3: DOALL I = 1,10
                S3
            ENDDO;
            L4: DOALL I = 1,1
                S4
            ENDDO
        PAREND
    END
PAREND
```

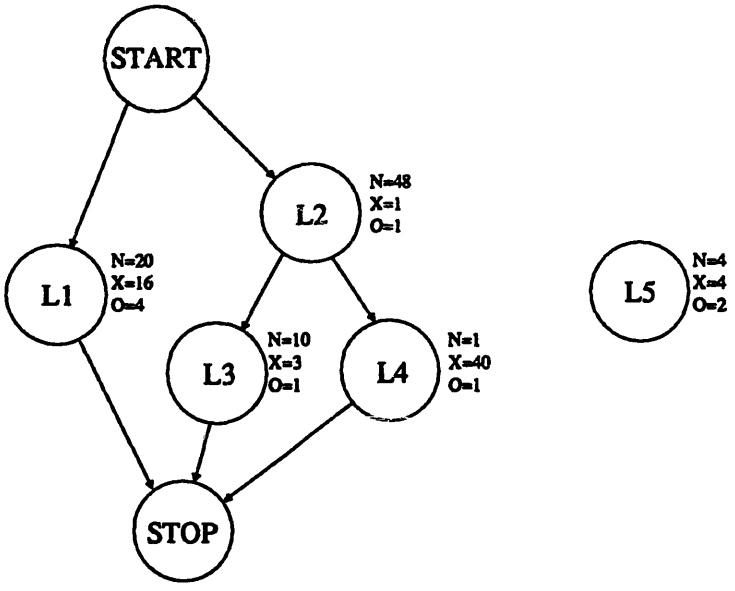

Fig. 2. An example program and its program graph.

entry point and one exit point. We shall use TASKBEGIN and TASKEND to denote the entry point and the exit point of a task.

Focusing on the partitioning of parallel loops, we may observe two types of parallelism: horizontal and vertical [16]. Horizontal parallelism results by simultaneously executing different iterations of the same parallel loop on different processors. Vertical parallelism, in turn, is the result of the simultaneous execution of two or more different loops. Now consider the following parallel loop:

```
L: DOALL I = 1, N
       B
ENDDO
```

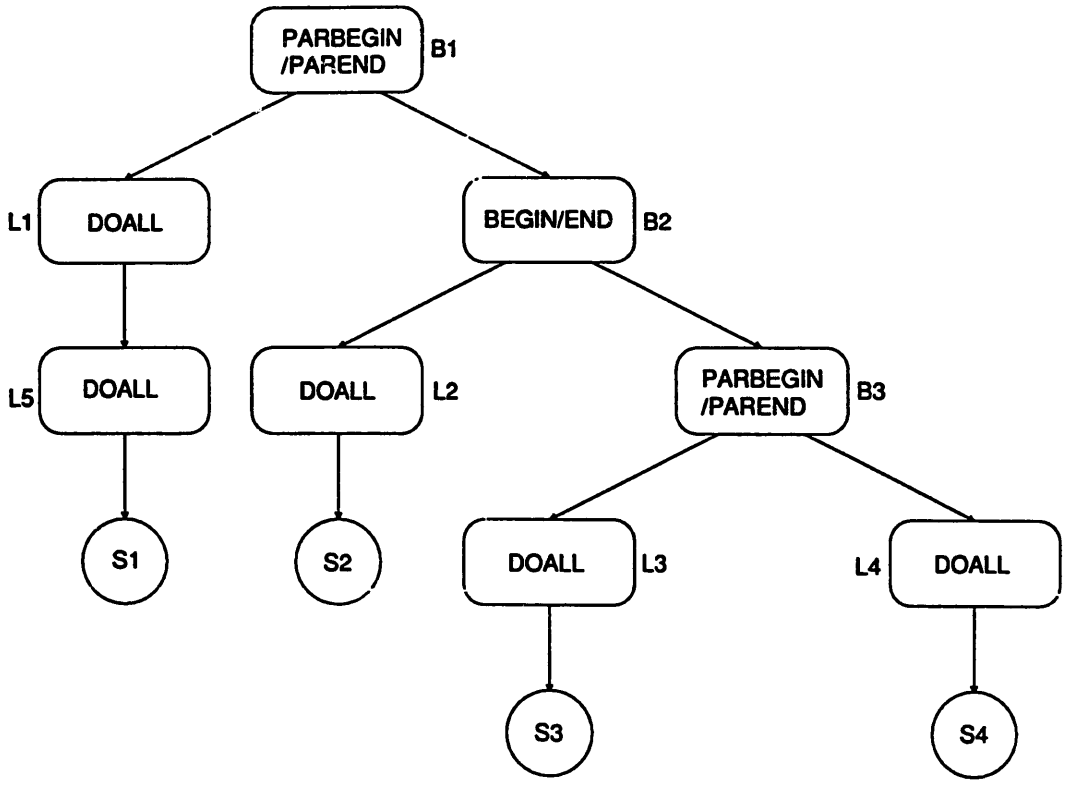

Fig. 3. The parse tree of the example program.

If this loop is partitioned into K tasks, the partitioned loop can be expressed as follows:

```
L: DOALL T = 1, K
     TASKBEGIN
          DOSER I = 1, N, K
               B
          ENDDO
     TASKEND
ENDDO
```

If a parallel loop L is partitioned as the above example, the partition of the loop L, denoted by Π_L, can be expressed as $\Pi_L = \{(L, K)\}$. A partition is structured if every task in this partition is obtained by partitioning some parallel node as the above example. Those parallel nodes partitioned are called primitive nodes. A structured partition of a parallel program can be defined as the union of partitions of primitive loops. Due to the restriction of functional tasks, parallelism between primitive nodes will be fully exploited while parallelism within a task will not be exploited. Therefore, in a structured partition, the precedence between tasks are the same as the precedence relations between primitive nodes. As an illustration, Fig. 4 and Fig. 5 give two structured partitions. Fig. 4 is an example partition that exploits all parallelism. Fig. 5 is the optimal partition generated by the algorithm proposed in Section 4.

2.4 The scheduling mechanism

The scheduling mechanism can have severe impacts on the performance of the multiprocessor systems. The general scheduling problem has been proved to be NP-complete in the strong sense [8]. It has been shown that, in certain case of random task graphs, optimal schedules can be achieved by deliberately keeping one or more processors idle in order to better

```
PARBEGIN
    DOALL K = 1,20
        DOALL J = 1,4
            TASKBEGIN
                S1
            TASKEND
        ENDDO
    ENDDO;
    BEGIN
        DOALL K = 1,48
            TASKBEGIN
                S2
            TASKEND
        ENDDO;
        PARBEGIN
            DOALL I = 1,10
                TASKBEGIN
                    S3
                TASKEND
            ENDDO;
            DOALL I = 1,1
                TASKBEGIN
                    S4
                TASKEND
            ENDDO
        PAREND
    END
PAREND
```

cost = 196.5 and crit = 43

Fig. 4. An example partition when $P = 4$.

utilize them in a later point. Detecting such anomalies, however, requires processing of the entire task graph in advance. Since this is not possible at run time, the luxury of deliberately keeping processors idle should not be permitted.

Therefore, we assume the following list scheduling algorithm is used. For convenience, we define a macro-actor to be a dynamic invocation of a static task. The distinction between a macro-actor and a task is akin to the distinction between a process and a program. In the run-time scheduling model, each processor repeats the following steps serially and there is no opportunity for overlapping communication with computation:

1. pick a ready macro-actor
2. fetch the macro-actor's inputs
3. execute the macro-actor
4. store the macro-actor's outputs.

An important property of this scheduling model and other list scheduling algorithms is that there is no intentionally introduced idleness. This means that a processor never stay idle if there is a macro-actor ready for execution. Despite of this limitation, this scheduling model is not an obstacle to achieve linear speedup because it was proved [7] that the schedule generated by a list scheduling algorithm have a parallel execution time at most twice the optimal parallel execution time.

```
PARBEGIN
    DOALL K = 1,7
        TASKBEGIN
            DOSEP I = K,20,7
                DOSER J = 1,4
                    S1
                ENDDO
            ENDDO
        TASKEND
    ENDDO;
    BEGIN
        DOALL K = 1,5
            TASKBEGIN
                DOSER I = K,48,5
                    S2
                ENDDO
            TASKEND
        ENDDO;
        PARBEGIN
            TASKBEGIN
                DOSER I = 1,10
                    S3
                ENDDO;
            TASKEND
            TASKBEGIN
                DOALL I = 1,1
                    S4
                ENDDO
            TASKEND
        PAREND
    END
PAREND
```

cost = 157.25 and crit = 52

Fig. 5. The optimal partition when $P = 4$.

3. Problem

For convenience, we define some notations to be used. Consider the execution of a program graph G under the partition Π for a single set of inputs. TOTAL(G, Π) is defined to be the total execution time of all macro-actors. CRIT(G, Π) is defined as the critical path execution time of the run-time precedence graph, i.e. the total execution time of those macro-actors that lie on the critical path of the run-time precedence graph. Therefore, CRIT(G, Π) is the parallel execution time of the program on infinite number of processors. Let $T_{par}(P)$ denote the parallel execution time of the program on P processors under the run-time scheduling model described in the previous section. Then, $T_{par}(P)$ approaches CRIT(G, Π) as the number of processors approaches infinity.

The goal of program partitioning is to find the optimal partition which minimizes parallel execution time with limited number of processors in the presence of overhead. Since the scheduling of tasks is performed at run time, there is no way to know the parallel execution time of a partitioned program at compile time. Fortunately, based on the scheduling model defined in the previous section, a theorem presented by Sarkar [18] provides us a reasonable

cost function to evaluate the performance of a partitioned program. This theorem is stated below without proof.

Theorem 3.1.

$$\frac{P-1}{2P}\text{CRIT}(G, \Pi) + \frac{1}{2P}\text{TOTAL}(G, \Pi) \leqslant T_{\text{par}}(P) \leqslant \frac{P-1}{P}\text{CRIT}(G, \Pi) + \frac{1}{P}\text{TOTAL}(G, \Pi).$$

Although different scheduling algorithms may generate different values of $T_{\text{par}}(P)$, Theorem 3.1 provides bounds on $T_{\text{par}}(P)$. Since the problem of finding a schedule with the smallest $T_{\text{par}}(P)$ is NP-complete [8], it is reasonable to use the worst-case performance instead of $T_{\text{par}}(P)$ as the cost function. Therefore we define the Structured Partition Problem and the Simplified Structured Partition Problem as follows.

Definition. Let $f(\Pi) = \frac{P-1}{P}\text{CRIT}(G, \Pi) + \frac{1}{P}\text{TOTAL}(G, \Pi)$ be the cost of partition Π.

Definition. The Structured Partition Problem is defined as follows. Given a program graph $G=(V, A)$ and a multiprocessor model $M=(P, R, W, S, E)$ as defined in the previous section, find a structured partition Π of the program graph G such that $f(\Pi)$ is minimized.

Definition. The Simplified Structured Partitioned Problem is defined as follows. Given an simple program graph $G=(V, A)$ and a multiprocessor model $M=(P, R, W, S, E)$, find a structured partition Π of the program graph G such that $f(\Pi)$ is minimized.

Note that, for a simple program graph, there are only two types of nodes: simple nodes and parallel nodes. Since a simple node can be treated as a parallel node with only one iteration, the processing of a simple node is the same as that of a parallel node. Therefore, there are only three types of internal nodes in the parse tree of a simple program graph. They are the PARBEGIN/PAREND nodes, the BEGIN/END nodes, and the DOALL nodes. Furthermore, every DOALL node is a primitive node for a simple program graph since there is no nested loops.

4. The prune-and-search algorithm

In this section, we consider the Simplified Structured Partition Problem first and propose an algorithm to solve this problem. The algorithm is then extended to solve the Structured Partition Problem. Like the definitions of the total execution time and the critical path execution time of a program in the previous section, we can define the total execution time and the critical path execution time of a node in the parse tree in a similar manner. In the following we shall discuss the computation of the total execution time and the critical path execution time of an internal node in an bottom-up manner. First, consider parallel node V_i partitioned into K_i tasks. For a parallel node V_i, we use the following notations: The partition of this parallel node is expressed as $\Pi_{V_i} = \{(V_i, K_i)\}$. N_i denotes the number of iterations of V_i. R_i denotes the time to read input data of an iteration of V_i from the shared memory modules and W_i denotes the time to write the output data of an iteration of V_i to the shared memory modules. CR_i denotes the time to read the input data that are common for all iterations of V_i (e.g. scalar variables). E_i denotes the time to perform the computation of an iteration of V_i. The

scheduling overhead incurred by scheduling a task of V_i is denoted by S_i. Note that some tasks of V_i will contain $\lceil N_i/K_i \rceil$ iterations while others contain $\lfloor N_i/K_i \rfloor - 1$ iterations. Since the input data common for all iterations of V_i can be read only once for each task, the total execution time and the critical path execution time of this parallel node can be expressed as the following equations:

$$\begin{aligned} \text{TOTAL}(V_i, \Pi_{\text{Vi}}) &= N_i * (E_i + R_i - \text{CR}_i + W_i) + K_i * (S_i + \text{CR}_i) \\ &= N_i * X_i + K_i * O_i \end{aligned} \tag{1}$$

$$\begin{aligned} \text{CRIT}(V_i, \Pi_{\text{Vi}}) &= \lceil N_i/K_i \rceil * (E_i + R_i - \text{CR}_i + W_i) + (S_i + \text{CR}_i) \\ &= \lceil N_i/K_i \rceil * X_i + O_i \end{aligned} \tag{2}$$

where $X_i = (E_i + R_i - \text{CR}_i + W_i)$ and $O_i = S_i + \text{CR}_i$.

Then consider a concurrent block of statements in the form of CB: PARBEGIN B_1; B_2; ...; B_n PAREND. By definition, the total execution time and the critical path execution time of this concurrent block can be computed as follows.

$$\text{TOTAL}(\text{CB}, \Pi_{\text{cb}}) = \sum_{i=1}^{n} \text{TOTAL}(B_i, \Pi_{\text{Bi}})$$

$$\text{CRIT}(\text{CB}, \Pi_{\text{cb}}) = \max_{i=1}^{n} \text{CRIT}(B_i, \Pi_{\text{Bi}})$$

where $\Pi_{\text{cb}} = \bigcup_{i=1}^{n} \Pi_{\text{Bi}}$.

Finally consider a sequential block of statements in the form of SB: BEGIN B_1; B_2; ...; B_n END. By definition, the total execution time and the critical path execution time of this sequential block can be computed as follows:

$$\text{TOTAL}(\text{SB}, \Pi_{\text{sb}}) = \sum_{i=1}^{n} \text{TOTAL}(B_i, \Pi_{\text{Bi}})$$

$$\text{CRIT}(\text{SB}, \Pi_{\text{sb}}) = \sum_{i=1}^{n} \text{CRIT}(B_i, \Pi_{\text{Bi}})$$

where $\Pi_{\text{cb}} = \bigcup_{i=1}^{n} \Pi_{\text{Bi}}$.

Since there are only these three types of internal nodes in the parse tree of a simple program graph, we can compute the total execution time and the critical path execution time of a simple program graph by applying the above equations in an bottom-up manner when the partition is given. However, there are so many possible partitions such that an exhaustive search of all possible partitions is impossible. In the following we try to find some ways to prune those partitions that, no matter how many processors are available, they are not capable of being the optimal partition. For example, let Π_1 and Π_2 be two possible partitions of the program graph G. If $\text{TOTAL}(G, \Pi_1) < \text{TOTAL}(G, \Pi_2)$ and $\text{CRIT}(G, \Pi_1) \leqslant \text{CRIT}(G, \Pi_2)$, we can conclude that, no matter how many processors are available, the partition Π_2 will not be the optimal partition of the program graph G because $f(\Pi_1) < f(\Pi_2)$. The following theorem is a generalization of this example.

Theorem 4.1. *Let Π_{opt} denote the optimal partition of the program graph G. If Π_1 and Π_2 are two possible partitions of some program segment B in the program graph G and* $\text{TOTAL}(B, \Pi_1) < \text{TOTAL}(B, \Pi_2)$ *and* $\text{CRIT}(B, \Pi_1) \leqslant \text{CRIT}(B, \Pi_2)$, *then* $\Pi_2 \not\subset \Pi_{\text{opt}}$.

Proof. Suppose $\Pi_2 \subset \Pi_{opt}$. Since $\text{TOTAL}(B, \Pi_1) < \text{TOTAL}(B, \Pi_2)$ and $\text{CRIT}(B, \Pi_1) \leqslant \text{CRIT}(B, \Pi_2)$, $f(G, \Pi_{opt} - \Pi_2 + \Pi_1) < f(G, \Pi_{opt})$. However, this contradicts the original assumption. Therefore $\Pi_2 \not\subset \Pi_{opt}$. This completes the proof of Theorem 4.1. □

Those partitions that can not be pruned by Theorem 4.1 are called candidate partitions. The optimal partition must be one of the candidate partitions. Given the number of processors available, we can find the optimal partition by searching all candidate partitions of the parallel program. Since the number of candidate partitions is far less than the number of possible partitions, the search of the optimal partition will become much faster. In the following discussion we shall assume the set of candidate partitions and their total execution times are stored in two arrays, called the candidate partition array and the candidate total-execution-time array, indexed by their critical path execution times. Let Ψ_B, CP_B and CT_B denote the set of candidate partitions, the candidate partition array and the candidate total-execution-time array of an internal node B, respectively. CP_B and CT_B are defined as follows:

$$\text{for all } \Pi \in \Psi_B\text{: } \text{CP}_B[\text{CRIT}(B, \Pi)] = \Pi.$$

$$\text{for all } \Pi \in \Psi_B\text{: } \text{CT}_B[\text{CRIT}(B, \Pi)] = \text{TOTAL}(B, \Pi).$$

As an illustration, consider the parallel loop $L3$ in the example program. All possible partitions of $L3$ is shown in Fig. 6(a). After applying Theorem 4.1 to this example, its candidate partition array and candidate total-execution-time array can be computed and are shown in Fig. 6(b). Note that $\text{CP}_B[C]$ and $\text{CT}_B[C]$ are undefined for some value C. In order to make the search of the optimal partition more easily, we shall use the extended candidate partition array ECP and the extended candidate total-execution-time array ECT in our algorithm. Fig. 6(c) shows the extended candidate partition array and the extended candidate total-execution-time array of $L3$. Formally, ECT_B is defined as follows:

$$\text{for all } C\text{: } \text{ECT}_B[C] = \min_{\substack{\Pi \\ \text{CRIT}(B, \Pi) \leqslant C}} \text{TOTAL}(B, \Pi).$$

For all C, $\text{ECP}_B[C] = \Pi$ where Π is the partition whose total execution time is minimum among all partitions that satisfy $\text{CRIT}(B, \Pi) \leqslant C$. In other words, $\text{ECP}_B[C] = \Pi$ if and only if $\text{ECT}_B[C] = \text{TOTAL}(B, \Pi)$. Note that the above definitions of ECP and ECT confirm with the definitions of CP and CT, i.e. for all $\Pi) \in \Psi_B$, $\text{ECP}_B[\text{CRIT}(B, \Pi)] = \Pi$ and $\text{ECT}_B[\text{CRIT}(B, \Pi)] = \text{TOTAL}(B, \Pi)$. An important property of ECT is that it is nonincreasing, i.e. $\text{ECT}[C_1] \geqslant \text{ECT}[C_2]$ if $C_1 < C_2$. In the following, we shall utilize this property to prune those partitions that are incapable of being the optimal partition and obtain the set of candidate partitions.

Now the problem is shifted to the computation of the extended candidate partition array and the extended candidate total-execution-time array of a parallel program. We shall compute these two array in an bottom-up manner. First consider a parallel node V_i. By definition, for all $C > X_i + O_i$:

$$\begin{aligned}
\text{ECT}_{Vi}[C] &= \min_{\substack{\Pi_{Vi} \\ \text{CRIT}(Vi, \Pi_{Vi}) \leqslant C}} \text{TOTAL}(V_i, \Pi_{Vi}) \\
&= \min_{\substack{1 \leqslant K_i \leqslant N_i \\ \lceil N_i/K_i \rceil * X_i + O_i \leqslant C}} (N_i * X_i + K_i * O_i) \\
&= N_i * X_i + \lceil N_i / \lfloor (C - O_i)/X_i \rfloor \rceil * O_i \qquad (3) \\
\text{ECP}_{Vi}[C] &= \{(V_i, \lceil N_i / \lfloor (C - O_i)/X_i \rfloor \rceil)\}. \qquad (4)
\end{aligned}$$

The consider a concurrent block of statements. Utilizing the nonincreasing property of the the extended candidate total-execution-time array, we can compute the extended candidate partition array and the extended total-execution-time array of a concurrent block based on the following two theorems.

Theorem 4.2. *For a concurrent block of statements in the form of* CB: PARBEGIN S_1; S_2 PARENDD, *the following equations hold*:

$$\textit{for all } C\text{: } \mathrm{ECT}_{cb}[C] = \mathrm{ECT}_{S1}[C] + \mathrm{ECT}_{S2}[C]. \tag{5}$$

$$\textit{for all } C\text{: } \mathrm{ECP}_{cb}[C] = \mathrm{ECP}_{S1}[C] \cup \mathrm{ECT}_{S2}[C]. \tag{6}$$

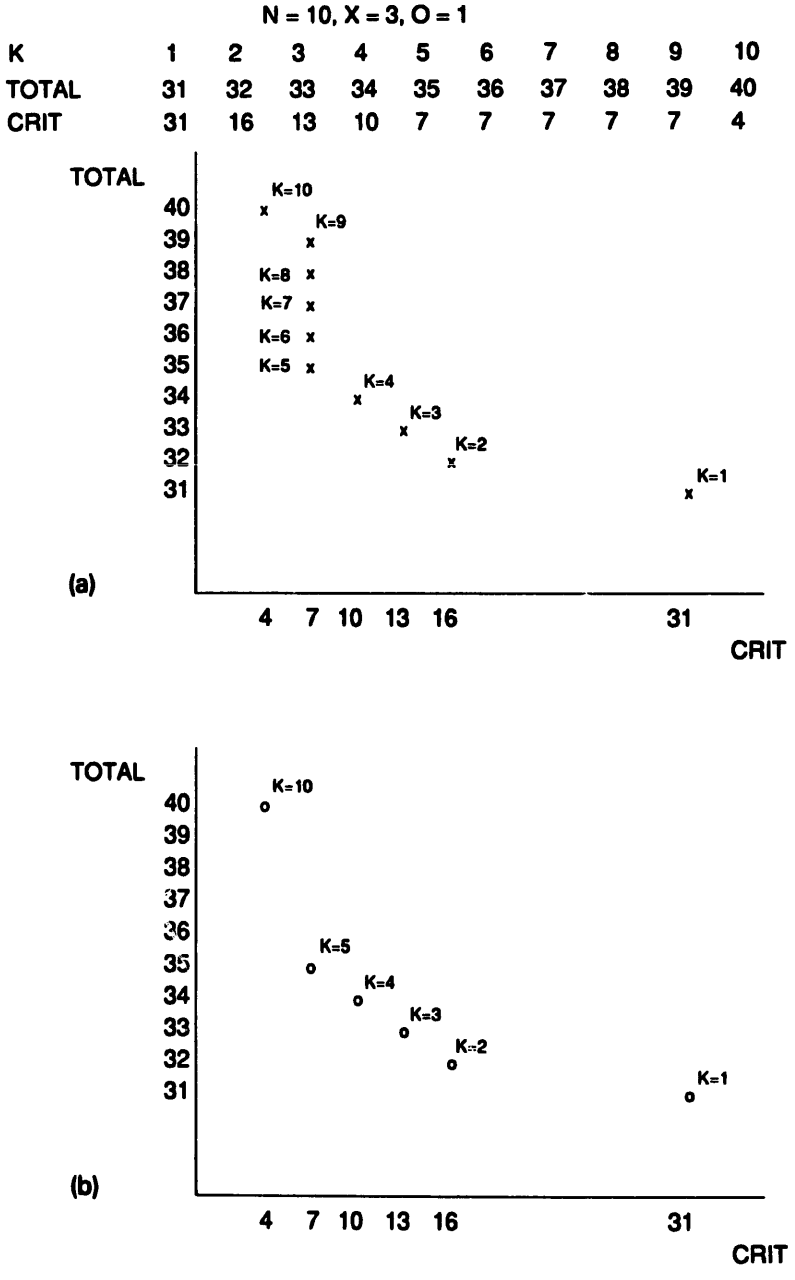

Fig. 6. (a) All possible partitions of the loop L3. (b) All candidate partitions of the loop L3. (c) The extended candidate partitions of the loop L3.

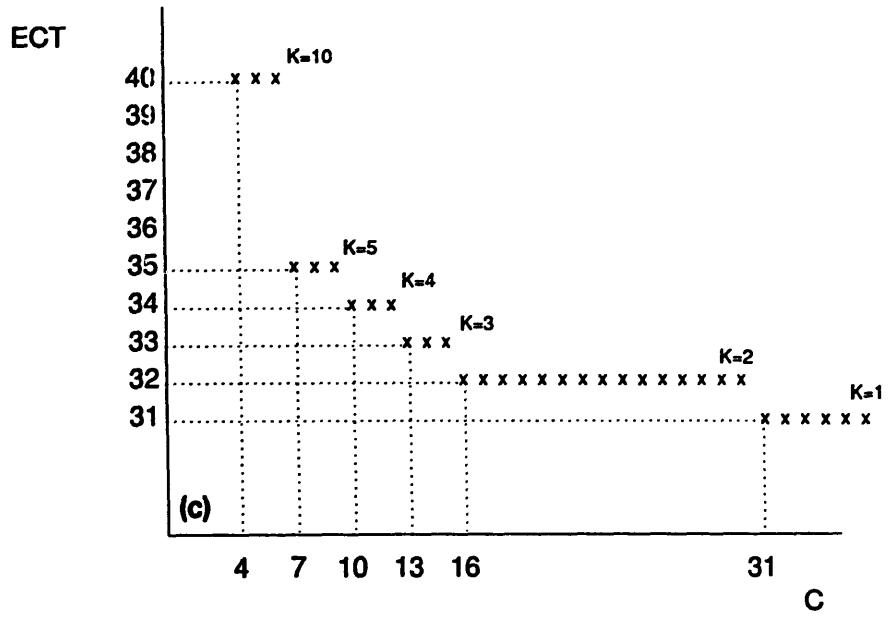

Fig. 6 (continued).

Proof. Assume that $ECT_{cb}[C] < ECT_{S1}[C] + ECT_{S2}[C]$ for some C. There must exist C_1 and C_2 such that $ECT_{cb}[C] = ECT_{S1}[C_1] + ECT_{S2}[C_2] < ECT_{S1}[C] + ECT_{S2}[C]$ and $\max(C_1, C_2) \leqslant C$. Then both $C_1 \leqslant C$ and $C_2 \leqslant C$ must hold. Since ECT_{S1} and ECT_{S2} are nonincreasing, both $ECT_{S1}[C_1] \geqslant ECT_{S1}[C]$ and $ECT_{S2}[C_2] \geqslant ECT_{S2}[C]$ must hold. Hence $ECT_{cb}[C] = ECT_{S1}[C_1] + ECT_{S2}[C_2] \geqslant ECT_{S1}[C] + ECT_{S2}[C]$. However, this contradicts the original assumption. Therefore Eq. (5) must hold and we can derive Eq. (6) accordingly. This completes the proof of Theorem 4.2. □

Theorem 4.3. *For a concurrent block of statements in the form of* CB: PARBEGIN S_1; S_2; ...; S_m PAREND, $m \geqslant 2$, *the following equations hold*:

$$\textit{for all } C\text{: } ECT_{cb}[C] = \sum_{i=1}^{m} ECT_{Si}[C] \tag{7}$$

$$\textit{for all } C\text{: } ECP_{cb}[C] = \bigcup_{i=1}^{m} ECP_{Si}[C]. \tag{8}$$

Proof. We shall prove this theorem by mathematical induction. In case of $m = 2$, it can be proved directly from Theorem 4.2. Assume that this theorem is valid when $m = n - 1$. Now consider the case that $m = n$. Since PARBEGIN S_1; ...; S_{n-1}; S_n PAREND is equivalent to PARBEGIN PARBEGIN S_1; ...; S_{n-1} PAREND; S_n PAREND, we can derive the following equations:

$$\textit{for all } C\text{: } ECT_{cb}[C] = \left(\sum_{i=1}^{n-1} ECT_{Si}[C]\right) + ECT_{Sn}[C] = \sum_{i=1}^{n} ECT_{Si}[C]$$

$$\textit{for all } C\text{: } ECP_{cb}[C] = \left(\bigcup_{i=1}^{n-1} ECP_{Si}[C]\right) \cup ECP_{Sn}[C] = \bigcup_{i=1}^{n} ECP_{Si}[C].$$

Thus Eqs. (7) and (8) are valid for $m = n$. By mathematical induction, this theorem is valid for all $m \geqslant 2$. □

Finally consider a sequential block of statements. Utilizing the nonincreasing property of the the extended candidate total-execution-time array, we can compute the extended candidate partition array and the extended total-execution-time array of a sequential block based on the following theorem.

Theorem 4.4. *For a sequential block of statements in the form of* SB: BEGIN S_1; S_2 END, *the following equations hold.*

$$\text{for all } C\text{: } \mathrm{ECT}_{sb}[C] = \min_{C'}(\mathrm{ECT}_{S1}[C'] + \mathrm{ECT}_{S2}[C - C']). \tag{9}$$

$$\text{for all } C\text{: } \mathrm{ECP}_{sb}[C] = \mathrm{ECP}_{S1}[C'] \cup \mathrm{ECP}_{S2}[C - C']$$
$$\text{if } \mathrm{ECT}_{sb}[C] = \mathrm{ECT}_{S1}[C'] + \mathrm{ECT}_{S2}[C - C'] \text{ for some } C'. \tag{10}$$

Proof. Assume that there exist C_1 and C_2 such that $C_1 + C_2 \leqslant C$ and

$$\mathrm{ECT}_{sb}[C] = \mathrm{ECT}_{S1}[C_1] + \mathrm{ECT}_{S2}[C_2] < \min_{C'}(\mathrm{ECT}_{S1}[C'] + \mathrm{ECT}_{S2}[C - C']).$$

Since ECT_{S2} is nonincreasing and $C_2 \leqslant C - C_1$, $\mathrm{ECT}_{S2}[C_2] \geqslant \mathrm{ECT}_{S2}[C - C_1]$. Therefore, $\mathrm{ECT}_{sb}[C] = \mathrm{ECT}_{S1}[C_1] + \mathrm{ECT}_{S2}[C_2] \geqslant \mathrm{ECT}_{S1}[C_1] + \mathrm{ECT}_{S2}[C - C_1]$. However, this contradicts the original assumption. Hence Eq. (9) must hold and we can derive Eq. (10) accordingly. □

Now consider a sequential block of statements, BEGIN S_1; S_2; ...; S_m END, $m \geqslant 2$. Since BEGIN S_1; S_2; ...; S_m END is equivalent to BEGIN BEGIN S_1; S_2 END; S_3; ...; S_m END, we can compute ECT_B and ECP_B of a sequential block by recursively applying the above transformation. From the above discussion, we can compute the extended candidate partition array and the extended candidate total-execution-time array of the program graph G in an bottom-up manner according to the parse tree of the given program. If the critical path execution time of the optimal partition is C_{opt}, then by definition the total execution time of the optimal partition must be $\mathrm{ECT}_G[C_{opt}]$ and the optimal partition must be $\mathrm{ECP}_G[C_{opt}]$. Thus by searching for the critical path execution time that minimizes the cost function, the optimal critical path execution time and the optimal partition of the given program can be obtained. The algorithm is shown in the follows.

Algorithm.

1. Compute the extended candidate partition array and the extended candidate total-execution-time array of each parallel node using Eqs. (3) and (4).
2. Compute the extended candidate partition array and the extended candidate total-execution-time array of any internal node in the parse tree of the program graph in an bottom-up manner using Eqs. (7) ~ (10).
3. Searching for the optimal critical path execution time C_{opt} such that the cost is minimized. The optimal partition Π_{opt} of the program graph G is $\mathrm{ECP}_G(C_{opt})$.

Now we shall estimate the time complexity of the above algorithm. Note that, for any internal node B, $\mathrm{ECT}_B[C]$ and $\mathrm{ECP}_B[C]$ is useless if $C > C_{opt}$. Therefore, it is unnecessary to compute ECT_B and ECP_B for all C. Let N denote the number of nodes in the program graph and C be an integer greater than or equal to C_{opt}. Obviously, the time complexity of the first step and the third step is $O(N)$ and $O(C)$, respectively. The second step takes $O(C)$ execution time for each concurrent block containing two statements and $O(C^2)$ execution time for each sequential block containing two statements. Therefore the second step takes $O(NC^2)$ execution time in the worst case. Since the most time-consuming step is the second step, the time complexity of the above algorithm is $O(NC^2)$.

In order to extend the above algorithm to solve the Structured Partition Problem, we must find some way to compute the extended candidate partition array and the extended candidate total-execution-time array of a nested loop. Now consider a parallel node V_i containing the program segment B as its loop body. When the parallel node V_i is a primitive node, its extended candidate total-execution-time array and extended candidate partition array can be compute as Eqs. (3) and (4). When the parallel node V_i is not a primitive node, its extended candidate total-execution-time array and extended candidate partition array can be computed from the extended total-execution-time array and the extended candidate partition array of the program segment B. Combining these two cases, we can derive the following equations.

$$\mathrm{ECT}_{V_i}[C] = \min\left(N_i * \mathrm{ECT}_B[C],\ N_i * X_i + \left\lceil N_i / \left\lfloor (C - O_i)/X_i \right\rfloor \right\rceil * O_i\right) \tag{11}$$

$$\mathrm{ECP}_{V_i}[C] = \begin{cases} \mathrm{ECP}_B[C], \text{ if } \mathrm{ECT}_{V_i}[C] = N_i * \mathrm{ECT}_B[C]. \\ \left\{\left(V_i, \left\lceil N_i / \left\lfloor (C - O_i)/X_i \right\rfloor \right\rceil\right)\right\}, \text{ otherwise.} \end{cases} \tag{12}$$

By recursively applying Eqs. (11) and (12) in addition to Eqs. (7) ~ (10), we can compute the extended candidate total-execution-time array and the extended candidate partition array of any program graph in a bottom-up manner according to the parse tree. Therefore we can solve the Structured Partition Problem in the same way as the Simplified Structured Partition Problem. Furthermore, it is obvious that the time complexity of the extended algorithm is the same as the original algorithm.

5. The linear time algorithm

Although the algorithm proposed in the previous section can generate the optimal partition, the computation time may be intolerable for some applications. In this section we proposed a simple algorithm that can generate a near optimal partition for a simple program graph in linear time. We shall prove that the cost of the partition generated by this linear time algorithm is at most twice the cost of the optimal partition. Therefore, the linear time algorithm can be used instead of the prune-and-search algorithm for the Simplified Structured Partition Problem when a near optimal partition can meet the requirement.

Recall that every DOALL node is a primitive node for a simple program graph, we need only to determine the number of tasks of each DOALL node. Now consider a DOALL node V_i partitioned into K_i tasks. According to Eqs. (1) and (2), we known that the total execution time is minimized when the number of tasks is one and the critical path execution time is minimized when the number of tasks is N_i. Let the minimum total execution time and the minimum critical path execution time of the parallel node V_i be expressed as MINTOTAL(V_i) and MINCRIT(V_i), then we can derive the following equations.

$$\mathrm{MINTOTAL}(V_i) = N_i * X_i + O_i \tag{13}$$

$$\mathrm{MINCRIT}(V_i) = X_i + O_i. \tag{14}$$

Suppose that $X_i < O_i$ and $K_i = 1 + \lfloor N_i * X_i / O_i \rfloor$, then we can derive the following three inequalities:

$$K_i = 1 + \lfloor N_i * X_i / O_i \rfloor \leqslant N_i$$

$$\begin{aligned} \mathrm{TOTAL}(V_i, \{(V_i, K_i)\}) &= N_i * X_i + (1 + \lfloor N_i * X_i / O_i \rfloor) * O_i \\ &\leqslant 2 * N_i * X_i + O_i \\ &\leqslant 2 * \mathrm{MINTOTAL}(V_i) \end{aligned}$$

$$\begin{aligned}\text{CRIT}(V_i, \{(V_i, K_i)\}) &= \lceil N_i/K_i \rceil * X_i + O_i \\ &\leqslant (N_i/K_i) * X_i + X_i + O_i \\ &\leqslant X_i + 2 * O_i \\ &\leqslant 2 * \text{MINCRIT}(V_i)\end{aligned}$$

On the other hand, suppose $X_i \geqslant O_i$ and $K_i = N_i$, then we can derive the following two inequalities:

$$\begin{aligned}\text{TOTAL}(V_i, \{(V_i, K_i)\}) &= N_i * X_i + N_i * O_i \\ &\leqslant 2 * N_i * X_i \\ &\leqslant 2 * \text{MINTOTAL}(V_i) \\ \text{CRIT}(V_i, \{(V_i, K_i)\}) &= \lceil N_i/N_i \rceil * X_i + O_i \\ &\leqslant X_i + O_i \\ &\leqslant 2 * \text{MINCRIT}(V_i)\end{aligned}$$

Note that $1 + \lfloor N_i * X_i/O_i \rfloor \leqslant N_i$ if and only if $X_i < O_i$. Thus if we partition the parallel node V_i into $K_i = \min(1 + \lfloor N_i * X_i/O_i \rfloor, N_i)$ tasks, the total execution time and the critical path execution time will be at most twice the minimum total execution time and the minimum critical path execution time, respectively. Accordingly, if we partition each parallel node V_i into $K_i = \min(1 + \lfloor N_i * X_i/O_i \rfloor, N_i)$ tasks, the total execution time and the critical path execution time of the partitioned program will be at most twice the total execution time and the critical path execution time of the optimal partition, respectively. In other words, the cost of the partitioned program is at most twice the cost of the optimal partition for a simple program graph. We can conclude that the cost of the partition generated by this method is close to the cost of the optimal partition.

A major advantage of this algorithm is that the time complexity of this algorithm is only $O(N)$ where N denotes the number of parallel nodes in the program graph. Hence we call this algorithm a linear time algorithm. Another advantage of this algorithm is that it can be used even when the numbers of iterations of parallel nodes are unknown and the cost is still near optimal. However, under this circumstance, the actual number of tasks of a parallel node is determined at the run time rather than at the compile time.

6. Conclusion

We have discussed the problem of partitioning concurrent programs into tasks for execution on multiprocessors. We have presented a program representation and a multiprocessor model for facilitating the partitioning work. By focusing our attention on the compile-time partitioning and run-time scheduling approach, we use an efficient list scheduling mechanism and provide a reasonable cost function to evaluate the cost of a partitioned program at compile time. Focusing on the partitioning of parallel loops, the structured partition and the Structured Partition Problem are defined.

Traditionally, heuristic algorithms are used to solve the partition problem. In stead of relying on heuristic algorithms, we base our research on mathematical modeling and present two algorithms. The first algorithms use a prune-and-search technique to reduce the time in searching the optimal partition for the Structured Partition Problem. The second algorithm will generate a near optimal partition for the Simplified Structured Partition Problem in linear time. We have proved that the partition generated by the linear time algorithm will have a cost at

most twice the cost of the optimal partition. Furthermore, this algorithm can be used even when the numbers of iterations of parallel nodes are unknown. Another advantage of our approach is that, unlike the heuristic approaches, we can analyze the performance of our approach according to the mathematical model.

References

[1] J.R. Allen and K. Kennedy, PFC: a program to convert Fortran to parallel form, in K. Hwang, ed., *Tutorial of Supercomputers: Design and Application* (IEEE, New York, 1984) 186–203.

[2] R.G. Babb, Parallel processing with large-grain data flow techniques, *Computer* (July 1984) 55–61.

[3] R.G. Babb, Programming the HEP with large-grain data flow techniques, in *Parallel MIMD Computation: HEP Supercomputer and Its Applications*, J.S. Kowalik, ed. (MIT Press, Cambridge, MA, 1985).

[4] R. Cytron, Limited processor scheduling of doacross loops, *Proc. 1987 Internat. Conf. Parallel Processing* (1987) 226–234.

[5] E.W. Dijkstra, Cooperating sequential processes, in *Programming Languages*, F. Genuys, ed. (Academic Press, New York, 1968) 43–112.

[6] Z. Fang, P.C. Yew, P. Tang and C.Q. Zhu, Dynamic processor self-scheduling for general parallel nested loops, *Proc. 1987 Internat. Conf. Parallel Processing* (1987) 1–10.

[7] R.L. Graham, Bounds on multiprocessing timing anomalies, *SIAM J. Appl. Math.* 17 (2) (1969).

[8] R.L. Graham, E.L. Lawler, J.K. Lenstra and A.H.G. Rinnooy Kan, Optimization and approximation in deterministic sequencing and scheduling: a survey, in *Annals of Discrete Mathematics* (North-Holland, Amsterdam, 1979) 287–326.

[9] R.W. Hockney and C.R. Jesshope, *Parallel Computers: Architecture, Programming, and Algorithms* (Adam Hilger, Bristol, 1981).

[10] K. Hwang and F.A. Briggs, *Computer Architecture and Parallel Processing* (McGraw-Hill, New York, 1984).

[11] D.J. Kuck, R.H. Kuhn, B. Leasure and M. Wolfe, The structure of an advanced vectorizer for pipelined processors, in *Proc. Fourth Internat. Comput. Software Applications Conf.* (Oct. 1980).

[12] D.J. Kuck et al, The effects of program restructuring, algorithm change and architecture choice on program performance, *Proc. 1984 Internat. Conf. Parallel Processing* (August, 1984).

[13] C. McCreary and H. Gill, Automatic determination of grain size for efficient parallel processing, *Comm. ACM* 32 (9) (1989) 1073–1078.

[14] K.J. Ottenstein, A brief survey of implicit parallelism detection, in *Parallel MIMD Computation: HEP Supercomputer and Its Applications*, J.S. Kowalik, ed. (MIT Press, Cambridge, MA, 1985).

[15] C.D. Polychronopoulos, D.J. Kuck and D.A. Padua, Utilizing multidimensional loop parallelism on large-scale parallel processor systems, *IEEE Trans. Comput.* C-38 (9) (1989) 1285–1296.

[16] C.D. Polychronopoulos and U. Banerjee, Processor allocation for horizontal and vertical parallelism and related speedup bounds, *IEEE Trans. Comput.* C-36 (4) (1987) 410–420.

[17] C.D. Polychronopoulos and D.J. Kuck, Guided self-scheduling: a practical scheduling scheme for parallel supercomputers, *IEEE Trans. Comput.* C-36 (12) (1987) 1425–1439.

[18] V. Sarkar, *Partitioning and Scheduling Parallel Program for Multiprocessors* (Pitman, London; also MIT Press, Cambridge, MA, 1989).

[19] H. Stone, Multiprocessor scheduling with the aid of network flow algorithms, *IEEE Trans. Software Eng.* SE-3 (1977).

[20] P. Tang and P.C. Yew, Processor self-scheduling for multiple-nested parallel loops, *Proc. 1986 Internat. Conf. Parallel Processing* (August, 1984) 528–535.

[21] M. Wolfe, Supercompilers for supercomputers, Ph. D. dissertation UIUCDCS-R-82-1105, Dep. Comput. Sci., Univ. Illinois, Urbana, 1982.